

Guida Completa: API REST e Cloud Computing

Introduzione

Questo documento costituisce una guida completa e dettagliata agli argomenti fondamentali di API REST e Cloud Computing, ideale per la preparazione a esami approfonditi. Troverai definizioni rigorose, spiegazioni dettagliate, esempi pratici e casi d'uso reali per ogni concetto.

PARTE 1: FONDAMENTI DI API

1.1 Definizione di API

Un'API (**Application Programming Interface**) è un'interfaccia che permette a due applicazioni software di comunicare tra loro in modo standardizzato e controllato. Rappresenta un insieme di regole, protocolli e strumenti che definiscono come i software devono interagire.

Caratteristiche fondamentali:

- Permette l'integrazione tra sistemi diversi
- Fornisce un'astrazione dei dettagli implementativi
- Garantisce coerenza e standardizzazione della comunicazione
- Espone solo le funzionalità necessarie (principio di information hiding)

Analogia quotidiana: Un'API è come il menu di un ristorante. Il client (cliente) non ha bisogno di sapere come il cuoco prepara il piatto (implementazione interna), ordina semplicemente dal menu (API) e riceve il risultato desiderato.

1.2 Differenti Tipi di API

API Locali (Desktop API):

- Comunicazione tra componenti dello stesso sistema operativo
- Esempio: API di Windows per accedere al file system
- Non richiedono rete

API Web (Web API):

- Basate su protocolli di rete (HTTP/HTTPS)
- Accessibili tramite Internet
- Permettono comunicazione client-server
- Esempio: API REST, GraphQL, SOAP

API Cloud:

- Servizi forniti da provider cloud

- Accessibili via Internet
- Basate su protocolli standard
- Esempio: AWS API, Azure API, Google Cloud API

API Interne (Private API):

- Utilizzate solo all'interno dell'organizzazione
- Non esposte pubblicamente
- Controllate direttamente dalla stessa azienda

API Pubbliche (Public API):

- Disponibili al pubblico
- Possono essere gratuite o a pagamento
- Documentate pubblicamente
- Esempio: API di Twitter, OpenWeather, GitHub

PARTE 2: API REST (Representational State Transfer)

2.1 Definizione e Principi Fondamentali

REST è uno stile architetturale per la progettazione di servizi web distribuiti. Non è uno standard formale, ma una collection di principi e constraints che guidano la costruzione di API web.

Definizione tecnica: REST è un approccio basato su risorse dove ogni entità è rappresentata da un URI (Uniform Resource Identifier) e le operazioni su tali risorse avvengono attraverso metodi HTTP standard.

Principi fondamentali di REST:

1. Client-Server Architecture

- Separazione chiara tra cliente (consumer) e server (provider)
- Indipendenza nell'evoluzione: cliente e server possono evolvere separatamente
- Comunicazione esclusivamente tramite interfaccia REST

2. Statelessness (Assenza di stato)

- Il server non mantiene informazioni sullo stato del client
- Ogni richiesta contiene tutte le informazioni necessarie per essere elaborata
- Aumenta la scalabilità e semplifica il design del server
- Esempio: Una richiesta GET /users/123 non dipende da richieste precedenti

3. Cachability

- Le risposte devono essere marcate come cacheable o non-cacheable
- Utilizza header HTTP come Cache-Control, ETag, Last-Modified
- Riduce il numero di richieste e migliora le prestazioni
- Esempio: Una GET su dati statici può essere cacheata, una POST generalmente no

4. Uniform Interface

- Interfaccia coerente e prevedibile
- Standardizzazione dell'accesso alle risorse
- Facilita la comprensione e l'uso dell'API

5. Layered System

- L'architettura può essere composta da strati multipli
- Client non conosce se è connesso direttamente al server finale
- Ogni strato ha responsabilità specifiche
- Migliora la scalabilità e la manutenibilità

6. Code on Demand (Facoltativo)

- Server può estendere le funzionalità del client inviando codice eseguibile
- Utilizzo limitato in pratica
- Esempio: JavaScript scaricato da un server web

2.2 Risorse e URI

Risorsa: Un'entità del dominio (utente, prodotto, ordine, commento) identificata univocamente e manipolabile tramite l'interfaccia.

URI (Uniform Resource Identifier): Una stringa che identifica univocamente una risorsa.

Come strutturare gli URI:

/api/v1/users → Collezione di utenti
 /api/v1/users/123 → Utente specifico con ID 123
 /api/v1/users/123/posts → Post dell'utente 123
 /api/v1/posts/456 → Post specifico con ID 456

Best practices per URI:

- Usare nomi plurali per collezioni: /users non /user
- Usare minuscole e trattini per separare parole: /user-profiles non /UserProfiles
- Evitare verbi negli URI: /users non /getUsers
- Usare versioning: /api/v1/, /api/v2/
- Mantenere una struttura logica e gerarchica

Identificativi univoci:

- Per risorse singole si usa solitamente l'ID numerico o UUID
- UUID (Universally Unique Identifier) è preferibile per alcuni casi
- Esempio: /api/v1/users/550e8400-e29b-41d4-a716-446655440000

2.3 Verbi HTTP (Metodi HTTP)

I verbi HTTP definiscono l'azione da eseguire sulla risorsa. Sono la base dell'interfaccia uniforme REST.

GET - Lettura (Read)

Descrizione: Richiede una rappresentazione di una risorsa. È il metodo più sicuro e idempotente.

Caratteristiche:

- Non modifica lo stato del server
- Idempotente: eseguire una GET 100 volte produce lo stesso risultato
- Cacheabile: il browser e i proxy possono cache le risposte GET
- Può avere un body nella richiesta, ma raramente usato

Esempi:

GET /api/v1/users → Restituisce lista di tutti gli utenti

GET /api/v1/users/123 → Restituisce i dettagli dell'utente 123

GET /api/v1/users?role=admin → Restituisce solo gli utenti admin (query params)

GET /api/v1/users?limit=10&offset=20 → Paginazione: 10 risultati a partire dal 20°

Risposta tipica:

Status: 200 OK

Content-Type: application/json

```
{  
  "id": 123,  
  "name": "Marco Rossi",  
  "email": "marco@example.com",  
  "role": "user"  
}
```

POST - Creazione (Create)

Descrizione: Crea una nuova risorsa. Non è idempotente: eseguire due POST identiche crea due risorse diverse.

Caratteristiche:

- Modifica lo stato del server (crea una nuova risorsa)
- Non idempotente: risultati diversi da esecuzioni ripetute
- Non cacheabile per definizione
- Può avere un body nella richiesta

Esempi:

POST /api/v1/users

Content-Type: application/json

```
{  
  "name": "Luca Bianchi",  
  "email": "luca@example.com",  
  "password": "securePassword123",  
  "role": "user"  
}
```

Risposta tipica:

Status: 201 Created

Location: /api/v1/users/124

Content-Type: application/json

```
{  
  "id": 124,  
  "name": "Luca Bianchi",  
  "email": "luca@example.com",  
  "role": "user"  
}
```

Note importanti:

- Restituisce generalmente 201 Created (non 200 OK)
- Dovrebbe restituire l'URI della risorsa appena creata nell'header Location
- Dovrebbe restituire la risorsa creata nel body

PUT - Aggiornamento Totale (Full Replace)

Descrizione: Sostituisce completamente una risorsa. È idempotente.

Caratteristiche:

- Modifica lo stato del server
- Idempotente: eseguire PUT 100 volte produce lo stesso risultato
- Sostituisce INTERAMENTE la risorsa (non è un merge)
- Richiede sempre un body

Esempi:

PUT /api/v1/users/123

Content-Type: application/json

```
{
  "name": "Marco Rossi Aggiornato",
  "email": "marco.nuovo@example.com",
  "role": "admin",
  "department": "IT"
}
```

Risposta tipica:

Status: 200 OK (o 204 No Content)

Content-Type: application/json

```
{
  "id": 123,
  "name": "Marco Rossi Aggiornato",
  "email": "marco.nuovo@example.com",
  "role": "admin",
  "department": "IT"
}
```

Differenza critica da POST:

- POST su /users crea una nuova risorsa con ID generato dal server
- PUT su /users/123 sostituisce l'utente 123 (il client conosce l'ID)

PATCH - Aggiornamento Parziale (Partial Update)

Descrizione: Aggiorna parzialmente una risorsa. Modifica solo i campi specificati.

Caratteristiche:

- Modifica solo i campi presenti nel body
- Non idempotente (secondo RFC 5789, ma in pratica spesso lo è)
- Più efficiente di PUT per aggiornamenti parziali
- Richiede un body

Esempi:

PATCH /api/v1/users/123

Content-Type: application/json

```
{  
  "email": "marco.nuovo@example.com"  
}
```

Dopo PATCH, l'utente avrà email aggiornata ma tutti gli altri campi rimangono uguali.

Risposta:

Status: 200 OK

Content-Type: application/json

```
{  
  "id": 123,  
  "name": "Marco Rossi",  
  "email": "marco.nuovo@example.com",  
  "role": "user"  
}
```

Confronto PUT vs PATCH:

PUT /api/v1/users/123 → Sostituisce TUTTO l'utente (richiede tutti i campi)

PATCH /api/v1/users/123 → Aggiorna solo i campi specificati

DELETE - Eliminazione (Delete)

Descrizione: Elimina una risorsa. È idempotente.

Caratteristiche:

- Modifica lo stato del server (elimina la risorsa)
- Idempotente: eliminare due volte la stessa risorsa produce lo stesso stato finale
- Generalmente non ha body nella richiesta
- Non cacheabile

Esempi:

DELETE /api/v1/users/123

Risposte tipiche:

Status: 204 No Content (senza body)

oppure

Status: 200 OK

Content-Type: application/json

```
{  
  "message": "Utente 123 eliminato con successo"  
}
```

HEAD e OPTIONS

HEAD:

- Identico a GET ma senza il body della risposta
- Usato per verificare se una risorsa esiste
- Riduce il traffico di rete

HEAD /api/v1/users/123

→ Restituisce gli header (incluso Content-Length) ma non il body

OPTIONS:

- Usato per scoprire quali metodi HTTP sono supportati da una risorsa
- Critico per CORS (Cross-Origin Resource Sharing)

OPTIONS /api/v1/users

→ Restituisce gli header che indicano quali verbi sono supportati

Allow: GET, POST, PUT, DELETE, PATCH, OPTIONS

2.4 Codici di Stato HTTP (Status Codes)

I codici di stato HTTP comunicano il risultato dell'operazione richiesta. Sono essenziali per una buona API REST.

Categorie di Status Code:

1xx - Informazioni (Informational)

- **100 Continue:** Continua a inviare il body della richiesta

2xx - Successo (Success)

- **200 OK:** La richiesta ha avuto successo. Usare per GET, PUT, PATCH, POST (quando non si crea una risorsa)
- **201 Created:** Una nuova risorsa è stata creata con successo. Usare per POST
- **202 Accepted:** La richiesta è stata accettata ma non ancora elaborata (per operazioni asincrone)
- **204 No Content:** La richiesta ha avuto successo ma non c'è contenuto da restituire. Usare per DELETE, PUT vuoto

3xx - Redirect

- **301 Moved Permanently:** La risorsa si è spostata in modo permanente a un nuovo URI
- **304 Not Modified:** La risorsa non è cambiata dal controllo precedente (caching)
- **307 Temporary Redirect:** La risorsa si è temporaneamente spostata

4xx - Errore Client

- **400 Bad Request:** La richiesta è malformata (parametri mancanti, JSON non valido)
- **401 Unauthorized:** Autenticazione richiesta o fallita
- **403 Forbidden:** Il client è autenticato ma non ha permessi per accedere (autorizzazione mancante)
- **404 Not Found:** La risorsa richiesta non esiste

- **405 Method Not Allowed:** Il metodo HTTP non è supportato per questa risorsa
- **409 Conflict:** La richiesta è in conflitto con lo stato attuale (es: duplicato)
- **422 Unprocessable Entity:** La richiesta è ben formata ma contiene errori semantici
- **429 Too Many Requests:** Troppi richieste in poco tempo (rate limiting)

5xx - Errore Server

- **500 Internal Server Error:** Errore generico del server
- **501 Not Implemented:** La funzionalità non è ancora implementata
- **503 Service Unavailable:** Il server è temporaneamente non disponibile

Uso corretto degli Status Code:

POST /api/v1/users → 201 Created (con Location header e body)

GET /api/v1/users/999 → 404 Not Found

PUT /api/v1/users/123 (dati invalidi) → 400 Bad Request

DELETE /api/v1/users/123 → 204 No Content

POST /api/v1/users (email duplicata) → 409 Conflict

GET /api/v1/users (non autenticato) → 401 Unauthorized

2.5 Content Negotiation

La negoziazione del contenuto permette al client di richiedere il formato desiderato per la risposta.

Accept Header (Client specifica cosa vuole):

GET /api/v1/users

Accept: application/json

→ Server restituisce JSON

Content-Type Header (Server specifica cosa invia):

200 OK

Content-Type: application/json

{...}

Formati comuni:

- application/json - JSON (standard di facto per le API REST)
- application/xml - XML
- text/plain - Testo puro
- text/html - HTML
- application/pdf - PDF

PARTE 3: LIVELLI DI MATURITÀ DELLE API (Richardson Maturity Model)

Leonard Richardson ha proposto un modello per valutare quanto un servizio web sia veramente "RESTful". Il modello ha 4 livelli (0-3), dove 3 rappresenta una vera API REST.

3.1 Livello 0: The Swamp of POX (Plain Old XML)

Caratteristica: Non è affatto REST. È semplicemente RPC (Remote Procedure Call) wrapped in HTTP.

Cosa manca:

- Nessuno sfruttamento dei metodi HTTP (usa solo POST)
- Nessuno sfruttamento dei status code HTTP
- Le "risorse" non sono identificate tramite URI
- Tutto è processi/azioni, non risorse

Esempi:

POST /api/doGetUser

Body: {userId: 123}

POST /api/doCreateUser

Body: {name: "Marco", email: "marco@example.com"}

POST /api/doDeleteUser

Body: {userId: 123}

Problema: Difficile da comprendere e mantenere. Non sfrutta gli standard HTTP.

3.2 Livello 1: Resources (Risorse)

Aggiunto rispetto al Livello 0:

- Introduzione delle risorse e dei loro URI univoci
- Ogni risorsa ha un endpoint dedicato
- Continua a usare solo POST per tutte le operazioni

Cosa manca ancora:

- Verbi HTTP diversi (ancora tutto POST)
- Status code HTTP appropriati

Esempi:

POST /api/users/123

Body: {action: "getDetails"}

POST /api/users

Body: {action: "create", name: "Marco", email: "marco@example.com"}

POST /api/users/123

Body: {action: "delete"}

Miglioramento: Almeno organizza il codice per risorse, ma non sfrutta HTTP.

3.3 Livello 2: HTTP Verbs (Verbi HTTP)

Aggiunto rispetto al Livello 1:

- Uso appropriato dei verbi HTTP (GET, POST, PUT, DELETE, PATCH)
- Uso appropriato dei status code HTTP
- Ogni risorsa ha metodi appropriati

Cosa manca ancora:

- HATEOAS (Hypermedia As The Engine Of Application State)
- Le risposte non contengono link per navigare le risorse correlate

Esempi:

GET /api/users/123 → 200 OK (oppure 404)

POST /api/users → 201 Created

PUT /api/users/123 → 200 OK (oppure 204 No Content)

PATCH /api/users/123 → 200 OK

DELETE /api/users/123 → 204 No Content (oppure 200 OK)

Questo è lo standard di facto oggi. La maggior parte delle API "REST" moderne si ferma al Livello 2. È un buon equilibrio tra semantica HTTP e praticità.

3.4 Livello 3: Hypermedia Controls (HATEOAS)

Aggiunto rispetto al Livello 2:

- Le risposte contengono link (hypermedia) per navigare le risorse correlate
- HATEOAS permette al client di scoprire dinamicamente le azioni disponibili
- Self-describing: la risposta include tutto ciò che serve per proseguire

Come funziona:

Richiesta:

GET /api/v1/users/123

Risposta (Livello 2):

```
{
  "id": 123,
  "name": "Marco Rossi",
  "email": "marco@example.com"
}
```

Risposta (Livello 3 - HATEOAS):

```
{
  "id": 123,
  "name": "Marco Rossi",
  "email": "marco@example.com",
  "_links": {
    "self": {
      "href": "/api/v1/users/123"
    },
    "update": {
      "href": "/api/v1/users/123",

```

```
"method": "PUT"
},
"delete": {
  "href": "/api/v1/users/123",
  "method": "DELETE"
},
"all_users": {
  "href": "/api/v1/users"
}
}
}
```

Vantaggi di HATEOAS:

- Il client non ha bisogno di conoscere l'intera struttura URI dell'API
- Se l'API cambia gli URI, il client continua a funzionare
- Documentazione implicita tramite i link disponibili
- Migliore versioning e evoluzione dell'API

Svantaggi:

- Aumenta la dimensione delle risposte
- Più complesso da implementare
- Non è uno standard formale (no standard format)
- Molti sviluppatori trovano superfluo

Standard per HATEOAS:

- **HAL (Hypertext Application Language):** Le risorse contengono una sezione `_links`
- **JSON-LD:** Usa JSON-LD per i link
- **Atom:** Format basato su XML

PARTE 4: CORS (Cross-Origin Resource Sharing)

4.1 Concetto di Origin

Un **origin** è una combinazione di:

- **Protocollo** (http, https)
- **Domain/Host** (example.com, localhost)
- **Porta** (80, 443, 3000, 8080)

Esempi di origin:

<https://example.com>

<https://example.com:8080>

<http://localhost:3000>

<https://api.example.com>

Same-Origin Policy:

La Same-Origin Policy è una politica di sicurezza fondamentale nei browser. Impedisce a JavaScript di una pagina di accedere ai dati di un'altra origin senza autorizzazione esplicita.

Esempio di violazione della SOP:

Pagina in: <https://example.com>

JavaScript prova ad accedere: <https://api.other-domain.com/users>

→ BLOCCATO dal browser (a meno che non ci sia CORS)

4.2 Cos'è CORS?

CORS (Cross-Origin Resource Sharing) è un meccanismo che permette ai server di indicare ai browser quali origin esterni possono accedere alle loro risorse.

Definizione tecnica: CORS è un protocollo basato su HTTP header che estende la Same-Origin Policy permettendo richieste cross-origin controllate e sicure.

Il problema che CORS risolve:

1. Un'applicazione web in <https://example.com> ha bisogno di dati da <https://api.other-domain.com>
2. Il browser blocca la richiesta per sicurezza (SOP)
3. CORS permette al server di dire "va bene, trust questa origin"

4.3 Richieste CORS: Simple vs Preflight

Simple CORS Requests

Una richiesta CORS è "semplice" se soddisfa TUTTI questi criteri:

Metodi consentiti:

- GET
- HEAD
- POST

Content-Type consentiti:

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

Headers consentiti:

- Accept
- Accept-Language
- Content-Language
- Content-Type (solo con i valori sopra)
- Otros: DPR, Dnlink, Save-Data, Viewport-Width, Width

Niente autenticazione:

- Nessun header Authorization
- Nessun cookie o credenziali

Esempio di Simple Request:

GET <https://api.other-domain.com/users>

Accept: application/json

Il browser invia automaticamente:

Origin: <https://example.com>

Server risponde con:

Access-Control-Allow-Origin: <https://example.com>

E il browser permette l'accesso al JavaScript.

Preflight CORS Requests

Se la richiesta NON soddisfa i criteri di "simple", il browser invia prima una richiesta **OPTIONS** di preflight.

Trigger del Preflight:

- Metodi: PUT, DELETE, PATCH, ecc.
- Content-Type: application/json, application/xml, ecc.
- Custom headers: Authorization, X-Custom-Header, ecc.
- Credentials (cookies, certificati HTTP)

Flow del Preflight:

1. Browser invia richiesta OPTIONS:
OPTIONS /api/v1/users
Origin: <https://example.com>
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Authorization, Content-Type
2. Server risponde:
HTTP/1.1 200 OK
Access-Control-Allow-Origin: <https://example.com>
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, PATCH
Access-Control-Allow-Headers: Authorization, Content-Type
Access-Control-Max-Age: 86400
3. Se il server approva, il browser invia la richiesta effettiva:
POST /api/v1/users
Authorization: Bearer token123
Content-Type: application/json
Origin: <https://example.com>

{...body...}

4. Server risponde normalmente (con gli stessi CORS header)

4.4 CORS Headers Principali

Request Headers (inviati dal browser):

- **Origin:** L'origin della pagina che fa la richiesta
- **Access-Control-Request-Method:** Per preflight, il metodo che si userà nella richiesta vera
- **Access-Control-Request-Headers:** Per preflight, gli header che si useranno nella richiesta vera

Response Headers (inviati dal server):

- **Access-Control-Allow-Origin:** Quale origin ha permesso di accedere
Access-Control-Allow-Origin: <https://example.com>
Access-Control-Allow-Origin: * (qualsiasi origin, meno sicuro)
- **Access-Control-Allow-Methods:** Quali metodi HTTP sono permessi
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, PATCH
- **Access-Control-Allow-Headers:** Quali header personalizzati sono permessi
Access-Control-Allow-Headers: Authorization, Content-Type, X-Custom-Header
- **Access-Control-Max-Age:** Quanti secondi cache il preflight (fino a 86400)
Access-Control-Max-Age: 3600 (1 ora)
- **Access-Control-Allow-Credentials:** Se i cookie/credenziali sono permessi
Access-Control-Allow-Credentials: true
Se true, Access-Control-Allow-Origin non può essere *
- **Access-Control-Expose-Headers:** Quali header della risposta il JavaScript può leggere
Access-Control-Expose-Headers: X-Total-Count, X-Page-Number

4.5 Implementazione CORS

Lato Server (Node.js/Express):

```
// Semplice CORS per tutti
const express = require('express');
const cors = require('cors');
const app = express();

app.use(cors());

app.get('/api/v1/users', (req, res) => {
  res.json([...]);
});
```

CORS selettivo (recommended):

```
const corsOptions = {
  origin: ['https://example.com', 'https://app.example.com'],
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH'],
  allowedHeaders: ['Content-Type', 'Authorization'],
  credentials: true,
  maxAge: 3600
};

app.use(cors(corsOptions));
```

CORS manuale (per controllo granulare):

```
app.use((req, res, next) => {
  const origin = req.headers.origin;
  const allowedOrigins = ['https://example.com', 'https://app.example.com'];

  if (allowedOrigins.includes(origin)) {
    res.setHeader('Access-Control-Allow-Origin', origin);
    res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH');
    res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');
    res.setHeader('Access-Control-Allow-Credentials', 'true');
  }
  next();
});
```

```
res.setHeader('Access-Control-Max-Age', '3600');
}

// Handle preflight
if (req.method === 'OPTIONS') {
  return res.sendStatus(200);
}

next();
});
```

Lato Client (JavaScript):

```
// Simple request
fetch('https://api.other-domain.com/users')
.then(res => res.json())
.then(data => console.log(data));

// Richiesta che trigga preflight
fetch('https://api.other-domain.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer token123'
  },
  credentials: 'include', // Includi cookie
  body: JSON.stringify({
    name: 'Marco',
    email: 'marco@example.com'
  })
})
.then(res => res.json())
.then(data => console.log(data));
```

4.6 CORS Errors e Troubleshooting

Errore comune:

Access to XMLHttpRequest at 'https://api.other-domain.com/users' from origin 'https://example.com' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

Significa: Il server non ha inviato gli header CORS appropriati.

Soluzioni:

1. Il server deve aggiungere CORS headers
 2. Passare attraverso un proxy che aggiunga i CORS headers
 3. Usare JSONP come alternativa (deprecato, non consigliato)
-

PARTE 5: CLOUD COMPUTING

5.1 Definizione di Cloud Computing

Cloud Computing è un modello di erogazione di servizi informatici in cui le risorse (storage, processamento, applicazioni, servizi) sono fornite tramite Internet e pagate su base pay-as-you-go.

Definizione NIST (National Institute of Standards and Technology):

Cloud Computing è un modello che consente accesso conveniente e on-demand a un pool condiviso di risorse informatiche configurabili (reti, server, storage, applicazioni, servizi) che possono essere rapidamente fornite e rilasciate con minimo sforzo di gestione o interazione con il provider.

Caratteristiche essenziali:

1. **On-Demand Self-Service:** L'utente può acquisire risorse automaticamente senza richiedere supporto umano
2. **Broad Network Access:** Accesso via Internet da diversi dispositivi
3. **Resource Pooling:** Le risorse sono condivise tra più utenti (multi-tenancy)
4. **Rapid Elasticity:** Le risorse possono essere aumentate o diminuite rapidamente
5. **Measured Service:** L'uso viene monitorato e fatturato (pay-as-you-go)

5.2 Modelli di Deployment del Cloud

Public Cloud

Definizione: Le risorse cloud sono possedute e gestite dal provider cloud, disponibili al pubblico tramite Internet.

Caratteristiche:

- Infrastruttura condivisa tra molte organizzazioni
- Massima scalabilità e flessibilità
- Costi più bassi (condivisione delle risorse)
- Minore controllo sulla sicurezza
- Conformità alle normative potenzialmente complessa

Esempi: AWS, Microsoft Azure, Google Cloud Platform, Heroku

Use case:

- Startup e PMI
- Applicazioni web pubbliche
- Carichi di lavoro variable
- Prototipazione rapida

Private Cloud

Definizione: Le risorse cloud sono possedute e gestite esclusivamente da un'organizzazione, non condivise con altri.

Caratteristiche:

- Infrastruttura dedicata ad una sola organizzazione
- Maggiore controllo e sicurezza
- Conformità alle normative più facile
- Costi iniziali elevati (investimento capex)
- Maggiore responsabilità di gestione e manutenzione

Esempi:

- OpenStack (software open-source per private cloud)
- VMware vCloud
- Enterprise cloud proprietari

Use case:

- Organizzazioni con requisiti di sicurezza elevati
- Dati sensibili o critici
- Industria finanziaria, sanitaria, governo
- Conformità normativa ristretta (GDPR, HIPAA)

Hybrid Cloud

Definizione: Combinazione di public e private cloud, con orchestrazione e standardizzazione tra i due.

Caratteristiche:

- Flessibilità nell'allocare i workload tra public e private
- Sicurezza dei dati sensibili in private cloud
- Scalabilità burst nel public cloud
- Complessità di gestione aumentata

Modelli di utilizzo:

- **Disaster Recovery:** Backup nel public cloud
- **Burst Capacity:** Workload normali in private, overflow nel public
- **Data Residency:** Dati sensibili in private, workload in public

Use case:

- Migrazione graduale al cloud
- Organizzazioni con requisiti misti di sicurezza
- Dev/Test in public cloud, Production in private cloud

Multi-Cloud

Definizione: Utilizzo di servizi da multiple provider cloud contemporaneamente.

Differenza da Hybrid: Multi-cloud usa multiple provider (AWS + Azure), hybrid usa public + private

Motivazioni:

- Evitare vendor lock-in
- Sfruttare servizi best-in-class di diversi provider
- Redundancy geografica

- Competizione tra provider per costi migliori

Sfide:

- Complessità di gestione
- Costi potenzialmente più alti
- Integrazione tra provider diverse

5.3 Modelli di Servizio del Cloud

Infrastructure as a Service (IaaS)

Definizione: Il provider fornisce infrastruttura virtualizzata (server, storage, networking) tramite Internet. L'utente gestisce OS, middleware, runtime, applicazioni, dati.

Cosa fornisce il provider:

- Server virtuali (VM)
- Storage (blocchi, file, object storage)
- Networking
- Virtualizzazione

Cosa gestisce l'utente:

- Applicazioni
- Dati
- Runtime
- Middleware
- Sistema Operativo
- Patch di sicurezza

Responsabilità condivisa (Shared Responsibility):

Provider: Infrastruttura fisica, virtualizzazione, rete

Utente: SO, applicazioni, dati

Esempi: AWS EC2, Microsoft Azure Virtual Machines, Google Compute Engine, DigitalOcean

Vantaggi:

- Massima flessibilità
- Scalabilità granulare
- Paghi solo quello che usi
- Nessun investimento capex

Svantaggi:

- Maggiore complessità gestionale
- Responsabilità sulla sicurezza dell'SO
- Aggiornamenti e patching a carico dell'utente

Use case:

- Applicazioni custom
- Ambienti di sviluppo/test
- Carichi di lavoro con requisiti particolari

- High-performance computing

Platform as a Service (PaaS)

Definizione: Il provider fornisce una piattaforma di sviluppo completa (middleware, runtime, linguaggi, database, strumenti). L'utente fornisce solo l'applicazione.

Cosa fornisce il provider:

- Runtime e middleware
- Linguaggi di programmazione
- Database
- Tools di sviluppo
- Framework
- Sistema Operativo
- Storage

Cosa gestisce l'utente:

- Applicazioni
- Dati

Responsabilità condivisa:

Provider: Infrastruttura, OS, runtime, database, middleware

Utente: Applicazioni, dati

Esempi: Heroku, Google App Engine, AWS Elastic Beanstalk, Salesforce, Firebase

Vantaggi:

- Sviluppo rapido
- Niente gestione infrastruttura
- Scalabilità automatica
- Deployment semplificato
- Built-in tools per security, monitoring

Svantaggi:

- Minore flessibilità rispetto a IaaS
- Vendor lock-in potenziale
- Costi possono essere elevati per usage elevato
- Limitazioni sul linguaggio/framework supportati

Use case:

- Startup e sviluppo rapido
- Applicazioni web standard
- Prototipi
- Applicazioni con picchi di carico variabili
- Team piccoli con risorse limitate

Software as a Service (SaaS)

Definizione: Il provider fornisce l'intera applicazione completa, accessibile via browser. L'utente non gestisce nulla tranne i propri dati.

Cosa fornisce il provider:

- Intera applicazione (UI, logica, database, infrastruttura)
- Aggiornamenti automatici
- Manutenzione
- Backup
- Sicurezza

Cosa gestisce l'utente:

- Dati inseriti
- Configurazione dell'applicazione

Responsabilità condivisa:

Provider: TUTTO (infrastruttura, applicazione, dati)

Utente: Configurazione, dati

Esempi: Office 365, Google Workspace, Salesforce, Slack, Zoom, GitHub, Jira Cloud, Dropbox

Vantaggi:

- Zero gestione infrastruttura
- Aggiornamenti automatici
- Accesso da qualsiasi dispositivo
- Collaborazione built-in
- Costi prevedibili (subscription)

Svantaggi:

- Nessun controllo sull'applicazione
- Customizzazione limitata
- Vendor lock-in significativo
- Dipendenza dalla connessione Internet
- Privacy/compliance concerns

Use case:

- Applicazioni office (email, collaborazione, productivity)
- CRM (Salesforce)
- Comunicazione (Slack, Teams)
- Storage e sync (Dropbox, OneDrive)
- Project management

Confronto IaaS vs PaaS vs SaaS

Responsabilità IaaS PaaS SaaS

Applicazione	Utente	Utente	Provider
Dati	Utente	Utente	Shared/Provider
Runtime/Middleware	Utente	Provider	Provider

SO Utente Provider Provider
Virtualizzazione Provider Provider Provider
Server fisici Provider Provider Provider
Storage fisico Provider Provider Provider
Rete Provider Provider Provider

Controllo Alto Medio Basso
Flessibilità Alta Media Bassa
Complessità gestionale Alta Media Bassa
Costi operativi Variabili Bassi Bassi
Vendor lock-in Basso Medio Alto

5.4 Vantaggi del Cloud Computing

1. Riduzione Costi (Cost Efficiency)

CapEx vs OpEx:

- **CapEx (Capital Expenditure):** Investimento in hardware fisico (upfront cost)
- **OpEx (Operational Expenditure):** Pagamento basato su utilizzo (pay-as-you-go)

Nel cloud passi da CapEx (costoso inizialmente) a OpEx (paghi quello che usi).

Risparmi:

- No acquisto hardware costoso
- No manutenzione hardware
- No cooling/electricity per server on-premise
- Scalabilità immediata senza over-provisioning

2. Scalabilità e Elasticità

Scalabilità: Capacità di aumentare risorse per gestire carico maggiore

Elasticità: Capacità di aumentare/diminuire risorse dinamicamente

Scenario on-premise:

- Carico massimo atteso: 1000 utenti simultanei
- Devi acquistare infrastruttura per 1000 utenti
- La maggior parte del tempo sottoutilizzato
- Peak inaspettato? Downtime

Scenario cloud:

- Scalabilità automatica: 100 → 1000 → 100 utenti
- Paghi solo il carico effettivo
- Peak affrontato automaticamente

3. Performance e Velocità

Deployment rapido:

- Infrastruttura pronta in minuti/ore
- No procurement, no installation
- Time-to-market ridotto

Global distribution:

- Servizi disponibili da data center sparsi globalmente
- Latenza ridotta (CDN edge locations)
- Disaster recovery geografico

4. Affidabilità (Reliability)

Alta disponibilità:

- Ridondanza automatica
- SLA (Service Level Agreement) tipicamente 99.9% - 99.99%
- Failover automatico
- Backup automatici

Disaster recovery integrato:

- Replicazione geografica
- Recovery point objective (RPO) bassissimo
- Recovery time objective (RTO) bassissimo

5. Sicurezza

Complice il reputation: I provider cloud investono massivamente in sicurezza

Vantaggi:

- Patching e aggiornamenti di sicurezza automatici
- Encryption (data at rest e in transit)
- Identity e access management robusti
- Conformità normativa (SOC 2, ISO 27001, GDPR, HIPAA)
- DDoS protection
- Monitoring 24/7

6. Flessibilità e Agilità

Evoluzione tecnologica rapida:

- Accesso a tecnologie nuove (AI, ML, Big Data) senza investimento
- Sperimentazione rapida
- Pivot tecnologico facile

Agile development:

- Ambienti di dev/test isolati
- DevOps facilitato
- CI/CD pipelines integrate

5.5 Sfide e Limitazioni del Cloud Computing

1. Vendor Lock-In

Problema: Una volta migrato su un provider cloud, cambiare provider è costoso e difficile

Cause:

- API proprietarie
- Servizi specifici non disponibili altrove
- Formato dati proprietari
- Training del team su specifico provider

Mitigazione:

- Architetture containerizzate (Docker, Kubernetes)
- Standardizzazione su open standards
- Evitare dipendenza da servizi proprietari dove possibile
- Multi-cloud strategy

2. Compliance e Normative

Sfide:

- Dati devono stare in specifiche giurisdizioni (GDPR, CCPA)
- Industrie regolate (finanza, sanità) hanno requisiti ristretti
- Audit e compliance complessi

Soluzioni:

- Cloud provider con regioni globali
- Private cloud per dati sensibili
- Encryption client-side
- Data residency agreements con provider

3. Performance e Latenza

Sfide:

- Applicazioni real-time sensibili alla latenza
- Carica di lavoro ad alte prestazioni può essere costosa
- Network latency unpredictable

Soluzioni:

- Edge computing (processamento sul dispositivo client)
- Cache locale
- CDN per content delivery
- Hybrid cloud (critical components on-premise)

4. Complessità Gestionale

Sfide:

- Multi-cloud requirements aggiungono complessità
- Monitoring di risorse sparse
- Ottimizzazione dei costi richiede expertise
- Skill gap nel team

Soluzioni:

- Tools di monitoring e orchestrazione (Terraform, Ansible, Kubernetes)
- Cloud cost optimization tools
- Training e hiring expertise

5. Sicurezza (il rovescio della medaglia)

Sfide:

- Fiducia nel provider
- Shared infrastructure (multi-tenancy risks)
- Accesso internet aumenta attack surface
- Data breach risks
- Insider threats del provider

Mitigazioni:

- Encryption end-to-end
- Network segmentation (VPCs)
- IAM granulare
- Audit logs e monitoring
- Scelta provider affidabile

6. Outages e Downtime

Rischio: Se il data center del provider ha problemi, i servizi sono down

Mitigazione:

- Multi-region deployment
- SLA che garantiscono availability
- Failover automatico
- Disaster recovery plan

5.6 Casi d'Uso del Cloud

Sviluppo e Test

Applicazione: Ambienti di development e staging

Vantaggi:

- Ambiente facilmente replicabile
- Scalabilità per testing
- No hardware fisico costoso
- Auto-scaling per load testing

Esempio: Team di developer usa AWS EC2 per dev, staging e testing

Backup e Disaster Recovery

Applicazione: Replica geografica di dati critici

Vantaggi:

- Costi inferiori rispetto a on-premise DR site
- Geo-redundancy automatica
- Recupero rapido (RTO/RPO bassi)

Esempio: On-premise production, AWS backup replicato cross-region

Big Data e Analytics

Applicazione: Processing di grandi volumi di dati

Vantaggi:

- Storage illimitato (S3, GCS)
- Processing power scalabile (Spark, Hadoop)
- Tools di analisi (Redshift, BigQuery, Looker)
- Costi per storage molto bassi

Esempio: Analisi dati da sensori IoT su Google Cloud BigQuery

Machine Learning e AI

Applicazione: Training e deployment di modelli ML

Vantaggi:

- GPU/TPU disponibili on-demand
- Servizi ML managed (TensorFlow, PyTorch)
- Feature store, model registry, serving
- Experimentation platform

Esempio: Training CNN per computer vision su AWS SageMaker

Web Application e Siti Statici

Applicazione: Hosting di applicazioni web

Vantaggi:

- CDN globale integrata
- Auto-scaling per traffic spikes
- SSL/TLS automatico
- Database managed

Esempio: React app su Vercel, backend Node.js su Heroku

IoT (Internet of Things)

Applicazione: Gestione di milioni di dispositivi IoT

Vantaggi:

- Messaging scalabile (MQTT, HTTP)
- Data ingestion massiva
- Processing in real-time
- Device management platform

Esempio: Sensori smart city che inviano dati ad Azure IoT Hub

5.7 Migrazione al Cloud

Strategie (5 Rs):

1. Rehost (Lift and Shift)

Migrazione diretta di applicazioni on-premise al cloud senza modifiche.

- Veloce
- Rischi minori
- Non sfrutta vantaggi cloud
- Esempio: Macchina virtuale on-premise → AWS EC2

2. Replatform (Lift, Tinker, and Shift)

Migrazione con piccole ottimizzazioni per il cloud.

- Benefici cloud significativi
- Minime modifiche di codice
- Esempio: App on [ASP.NET](#) → AWS RDS (managed database)

3. Refactor (Re-architect)

Riscrittura dell'applicazione per sfruttare appieno il cloud.

- Benefici massimi
- Microservices, serverless
- Investimento di tempo significativo
- Esempio: App monolitica → Microservices su Kubernetes

4. Repurchase (Drop and Shop)

Abbandonare l'applicazione attuale e adottare SaaS.

- Eliminazione costi di manutenzione
- Aggiornamenti automatici
- Vendor lock-in
- Esempio: Custom CRM → Salesforce

5. Retire (Decommission)

Eliminare applicazioni non più necessarie.

- Riduzione complessità
- Eliminazione costi
- Esempio: Sistemi legacy obsoleti

5.8 Cloud Providers Principali

Amazon Web Services (AWS)

Market share: ~32% (leader)

Servizi key:

- EC2 (compute)
- S3 (storage)
- RDS (database)
- Lambda (serverless)
- CloudFront (CDN)
- IAM (identity management)

Vantaggi:

- Ampio portfolio servizi
- Maturo e stabile
- Community grande
- Prezzi competitivi

Svantaggi:

- Complesso inizialmente
- Molti servizi (paralysis by choice)

Best for: Enterprise, startup in fase di crescita, case use specifiche

Microsoft Azure

Market share: ~23% (secondo)

Servizi key:

- Virtual Machines (compute)
- App Service (PaaS)
- Azure SQL (database)
- Azure Functions (serverless)
- Cosmos DB (NoSQL)
- Synapse (analytics)

Vantaggi:

- Integrazione con ecosistema Microsoft
- Buono per enterprise on Windows
- Competitive pricing

- Governance/compliance robusto

Best for: Enterprise Microsoft, hybrid cloud, compliance requirements

Google Cloud Platform (GCP)

Market share: ~10% (terzo)

Servizi key:

- Compute Engine (compute)
- Cloud Storage (storage)
- Cloud SQL (database)
- BigQuery (analytics)
- Cloud Run (serverless)
- Vertex AI (machine learning)

Vantaggi:

- Eccellente per Big Data/Analytics
- Leader in AI/ML
- Buon pricing
- Kubernetes native

Best for: Data analytics, ML/AI, startup tech-forward

Altre Piattaforme

Oracle Cloud:

- Forte su database
- Competitive per enterprise
- Crescita in corso

IBM Cloud:

- Hybrid cloud specializzata
- Valore per enterprise legacy

Heroku, DigitalOcean, Linode:

- PaaS più semplice
- Developer-friendly
- Startup/small projects

PARTE 6: CLOUD STORAGE

6.1 Tipi di Storage nel Cloud

Block Storage (Archiviazione a Blocchi)

Definizione: Storage organizzato in blocchi di dimensione fissa, come dischi rigidi tradizionali

Caratteristiche:

- Performance altissima
- Bassa latenza
- Usato per database, applicazioni transazionali
- Ogni block è identificato da indirizzo

Esempi:

AWS: EBS (Elastic Block Store)

Azure: Managed Disks

GCP: Persistent Disks

Use case:

- Database (MySQL, PostgreSQL)
- Transazioni high-speed
- Applicazioni che richiedono latenza minima

Vantaggi:

- Performance massima
- Controllo fine granularità
- Consistenza forte

Svantaggi:

- Costoso
- Meno scalabile per storage molto grande

File Storage (Archiviazione File)

Definizione: Storage organizzato in file system tradizionali con directory/cartelle

Caratteristiche:

- Accesso tramite protocolli NFS, SMB, CIFS
- Organizzazione gerarchica (directory tree)
- Permessi su file/folder
- Condivisione tra più istanze

Esempi:

AWS: EFS (Elastic File System), FSx

Azure: Azure Files

GCP: Cloud Filestore

Use case:

- Applicazioni che usano file system tradizionale
- Home directory per worker
- Shared storage tra VM

- NAS in cloud

Vantaggi:

- Familiare ai developer tradizionali
- Condivisione semplice tra istanze
- Permessi di file/folder

Svantaggi:

- Meno scalabile del object storage
- Performance inferiore al block storage

Object Storage (Archiviazione di Oggetti)

Definizione: Storage basato su oggetti (file) con metadata, non gerarchico

Caratteristiche:

- Ogni oggetto ha key univoca (es: s3://bucket/path/file.txt)
- Metadata richiedono query (no directory tree)
- Scalabilità praticamente infinita
- Durabilità molto elevata (11 nines - 99.999999999%)
- Costo bassissimo per GB
- Accesso via HTTP/REST API

Esempi:

AWS: S3 (Simple Storage Service) - gold standard

Azure: Blob Storage

GCP: Cloud Storage

Organizzazione:

Traditional: /documents/invoices/2024/january/invoice-001.pdf

S3: s3://my-bucket/documents/invoices/2024/january/invoice-001.pdf

Use case:

- Backup e archivio
- Hosting contenuto static (immagini, video)
- Data lake (big data)
- Log storage
- Backup database
- Disaster recovery
- CDN origin
- Data lakes

Vantaggi:

- Infinitamente scalabile
- Costo bassissimo
- Durabilità estrema
- Accesso globale
- Versioning automatico

Svantaggi:

- Latenza leggermente più alta del block storage
- No traditional file system operations (no mv, rm, mkdir)
- Transazioni eventually consistent (vs strong consistency del block)

Pricing Comparison:

Block Storage (EBS): ~\$0.10-\$0.15 per GB/mese

File Storage (EFS): ~\$0.30 per GB/mese

Object Storage (S3): ~\$0.023 per GB/mese (100x più economico!)

6.2 Cloud Database (Database Managed)

Relational Databases (SQL)

Definizione: Database che usano SQL, organizzati in tabelle con relazioni

Esempi di DB:

MySQL, PostgreSQL, Oracle, SQL Server, MariaDB

Servizi cloud managed:

AWS: RDS (Relational Database Service), Aurora

Azure: Azure SQL Database

GCP: Cloud SQL

Vantaggi managed:

- No gestione infrastruttura
- Backup automatici
- Replicazione e failover automatico
- Patching automatico
- Monitoring e alerting
- Read replicas per scalabilità lettura

Svantaggi:

- Meno controllo rispetto a IaaS
- Alcuni workaround non possibili
- Vendor lock-in

Use case:

- E-commerce (transazioni ACID)
- Applicazioni line-of-business
- CRM, ERP
- Any app che richiede relazioni complesse e transazioni

NoSQL Databases

Definizione: Database non relazionali, schema-less, ottimizzati per scalabilità orizzontale

Tipi:

Document Databases (es: MongoDB, Firestore):

// Documento (non tabella)

```
{
  "_id": "user_123",
```

```
"name": "Marco",  
"email": "marco@example.com",  
"addresses": [  
  { "street": "Via Roma", "city": "Milano" }  
]
```

Key-Value Stores (es: Redis, DynamoDB):

Key: user_123

Value: {"name": "Marco", "email": "marco@example.com"}

Column Family Stores (es: HBase, Cassandra):

Ottimizzati per query su colonne specifiche di milioni di righe

Graph Databases (es: Neo4j):

Ottimizzati per relazioni complesse (social networks, raccomandazioni)

Time-Series Databases (es: InfluxDB, CloudWatch):

Ottimizzati per dati con timestamp (metriche, monitoring)

Servizi cloud:

AWS: DynamoDB, DocumentDB, Neptune

Azure: Cosmos DB

GCP: Firestore, Datastore, BigTable

Vantaggi:

- Scalabilità orizzontale (distribuzione)
- Performance alta per letture massicce
- Schema flessibile
- Denormalizzazione naturale

Svantaggi:

- No ACID transazioni (in generale)
- Consistenza eventuale
- Complessità query complesse
- No join query native (generalmente)

Use case:

- Real-time analytics
- IoT data collection
- User profiles (flessibili)
- Cache (Redis)
- Social networks (graph)
- Time-series (monitoring, logs)
- High-volume writes

Data Warehousing

Definizione: Repository centralizzato di dati strutturati da molteplici sorgenti, ottimizzato per analytics

Caratteristiche:

- Schema fisso (star schema, snowflake schema)
- OLAP (Online Analytical Processing) - query analytics
- Columnar storage per performance
- No transazioni in tempo reale

Servizi cloud:

AWS: Redshift

Azure: Synapse

GCP: BigQuery

Snowflake: Data Cloud (multi-cloud)

Use case:

- Business Intelligence (BI)
- Analytics su dati storici
- Reporting centralizzato
- ML data preparation

6.3 Managed Kubernetes nel Cloud

Kubernetes: Piattaforma di orchestrazione container open-source

Servizi managed:

AWS: EKS (Elastic Kubernetes Service)

Azure: AKS (Azure Kubernetes Service)

GCP: GKE (Google Kubernetes Engine)

Vantaggi del managed:

- No gestione control plane
- Aggiornamenti automatici
- Scaling automatico
- Monitoring integrato
- Cost optimization

6.4 Serverless Computing

Definizione: Model dove il developer scrive solo funzioni, provider gestisce tutto il resto

Caratteristiche:

- Pay-per-execution (non per server)
- Auto-scaling implicito
- Niente infrast gestire
- Cold start latency (initialization delay)

Servizi:

AWS: Lambda

Azure: Azure Functions

GCP: Cloud Functions

Use case:

- Webhook handlers
- API lightweight
- Scheduled tasks (cron)
- Event-driven processing
- Microservices lightweight

Svantaggi:

- Cold start latency (ms - sec per prima esecuzione)
- Timeout limits
- Memory limits
- Vendor lock-in significativo
- Debugging difficile

PARTE 7: SECURITY NEL CLOUD

7.1 Principi di Security nel Cloud

Defense in Depth

Multipli strati di sicurezza:

Perimetrale (Firewall) → Network Segmentation → Application Security → Data Encryption

Least Privilege

Accesso minimo necessario:

User X ha permesso SOLO su /api/users/profile personale, non su altri users

Zero Trust

Assumere che tutti gli accessi sono potenzialmente compromessi:

Verificare OGNI richiesta

No "trusted network" - tutti gli accessi devono essere autenticati e autorizzati

7.2 Identity & Access Management (IAM)

Definizione: Controllo chi ha accesso a quali risorse e quali azioni può eseguire

Componenti:

Authentication: Verifica l'identità (chi sei?)

Username/password, 2FA, SSO, OAuth, JWT

Authorization: Verifica i permessi (cosa puoi fare?)

Role-based access control (RBAC)

Attribute-based access control (ABAC)

Policy-based access control

Implementazione nel cloud:

AWS IAM:

- Users, Groups, Roles
- Policy documents (JSON)
- MFA (Multi-Factor Authentication)

Azure AD/Entra ID:

- Users, Groups, Roles
- Conditional Access
- Risk detection

GCP IAM:

- Service accounts
- Roles (Basic, Predefined, Custom)
- Conditions

7.3 Network Security

VPC (Virtual Private Cloud)

Rete privata virtuale dove le risorse operano isolate

Componenti:

- **Subnet:** Divisione della VPC
- **Security Group / Network ACL:** Firewall
- **Route Tables:** Routing del traffico
- **VPN / Bastion Host:** Accesso a VPC da esterno

Segmentation:

Public Subnet: Load balancer, API gateway

Private Subnet: Applicazione server

Database Subnet: Database (accesso solo da app)

DDoS Protection

Protezione da attacchi distributed denial-of-service:

AWS Shield: Protezione base automatica

AWS WAF: Web Application Firewall

7.4 Data Security

Encryption at Rest

Dati criptati quando immagazzinati

Tipi:

- **Server-side:** Provider gestisce le chiavi (default S3, RDS)
- **Client-side:** Client cripta prima di inviare (massima privacy)
- **Customer-managed keys:** Client gestisce chiavi master (CMK)

Encryption in Transit

Dati criptati quando trasmessi su rete

Protocolli:

HTTPS/TLS: Per HTTP API

VPN: Per private network

TLS certificates: Per database connections

Key Management

Key Management Service (KMS):

AWS KMS, Azure Key Vault, GCP Cloud KMS

Gestisce encryption keys, audit logging, rotation automatica

7.5 Compliance e Auditing

Standards: SOC 2, ISO 27001, HIPAA, GDPR, PCI-DSS

Cloud provider audit:

CloudTrail (AWS): Logging di tutti gli API calls

Activity Log (Azure): Tutte le azioni

Cloud Audit Logs (GCP): API audit trails

PARTE 8: MONITORAGGIO E OBSERVABILITY

8.1 Monitoraggio (Monitoring)

Raccolta di metriche su sistema salute

Metriche key:

- **CPU utilization:** Carico di processo
- **Memory usage:** RAM consumata
- **Disk I/O:** Velocità lettura/scrittura
- **Network bandwidth:** Traffic
- **Application latency:** Response time
- **Request rate:** Richieste al secondo
- **Error rate:** % di richieste fallite

Tools:

AWS: CloudWatch

Azure: Azure Monitor

GCP: Cloud Monitoring

Alerting: Notifiche quando metrica supera soglia

IF `cpu_utilization > 80%` for 5 minutes

THEN `send_email(ops_team) + auto_scale_up()`

8.2 Logging (Logging)

Raccolta di log di applicazione e sistema

Tipi di log:

- **Application logs:** Print statements nel codice
- **System logs:** OS events
- **API logs:** Request/response
- **Access logs:** Chi ha acceduto a cosa
- **Audit logs:** Chi ha modificato cosa (compliance)

Aggregazione e Search:

AWS CloudWatch Logs, AWS CloudTrail

Azure Application Insights

GCP Cloud Logging

Stack: Elasticsearch + Logstash + Kibana (ELK)

Stack: Datadog, New Relic, Splunk

8.3 Distributed Tracing

Traccia richieste attraverso microservices

Flow:

User request → API Gateway (service A)

→ Service B (internal call)

→ Service C (database query)

→ Response back to user

Tools:

AWS X-Ray

Azure Application Insights

GCP Cloud Trace

OpenTelemetry (open source)

Jaeger (open source)

PARTE 9: BEST PRACTICES

9.1 API Best Practices

1. **Versionamento:** /api/v1/users non /users
2. **Rate limiting:** Protezione da abuse
3. **Caching:** Header Cache-Control appropriati
4. **Pagination:** Per collezioni grandi
5. **Validation:** Input validation severa
6. **Error handling:** Messaggi di errore coerenti e informativi
7. **Documentation:** OpenAPI/Swagger spec
8. **Security:** Authentication (JWT, OAuth), HTTPS, CORS appropriato

9.2 Cloud Best Practices

1. Cost optimization:

- Reserved instances per workload predicibile
- Spot instances per batch jobs
- Right-sizing (non over-provision)
- Monitor usage costantemente

2. Disaster recovery:

- Multi-region deployment
- Backup estratti dalla regione
- RPO/RTO defined

3. Auto-scaling:

- CPU/memory metrics per scale-up
- Cooldown period per evitare flapping
- Scale-down conservativo

4. Infrastructure as Code:

- Terraform, CloudFormation, ARM templates
- Version control
- Reproducibility

5. DevOps practices:

- CI/CD pipeline
- Automated testing
- Blue-green deployments
- Immutable infrastructure

6. Security posture:

- Regular security audits
- Penetration testing
- Vulnerability scanning
- Least privilege access

DOMANDE D'ESAME ESEMPIO

Domande a Scelta Multipla

1. Quale delle seguenti è una caratteristica essenziale del Cloud Computing secondo NIST?

- A) Proprietà dell'infrastruttura da parte dell'utente
- B) On-Demand Self-Service
- C) Nessun accesso via Internet
- D) Pagamento una tantum

Risposta: B

2. In quale livello di maturità API (Richardson Model) si usa correttamente i verbi HTTP?

- A) Livello 0
- B) Livello 1
- C) Livello 2
- D) Livello 3

Risposta: C

3. Quale status code HTTP dovresti usare quando crei una nuova risorsa con POST?

- A) 200 OK
- B) 201 Created
- C) 202 Accepted
- D) 204 No Content

Risposta: B

4. La Same-Origin Policy blocca una richiesta JavaScript da <https://example.com> a <https://api.other-domain.com>. Come si abilita?

- A) JSONP
- B) CORS
- C) HTTPS force
- D) JWT token

Risposta: B

5. In un preflight CORS request, quale metodo HTTP usa il browser automaticamente?

- A) GET
- B) POST
- C) OPTIONS
- D) HEAD

Risposta: C

6. Quale differenza principale tra PUT e PATCH?

- A) PUT è idempotente, PATCH no
- B) PUT sostituisce l'intera risorsa, PATCH modifica parzialmente
- C) PUT è solo per create, PATCH è solo per update
- D) Non c'è differenza

Risposta: B

7. Quale Cloud service model fornisce solo l'applicazione finita?

- A) IaaS
- B) PaaS
- C) SaaS
- D) DaaS

Risposta: C

8. In quale Cloud deployment la sicurezza è massima ma i costi iniziali sono elevati?

- A) Public Cloud
- B) Hybrid Cloud
- C) Private Cloud
- D) Community Cloud

Risposta: C

9. Quale tipo di Cloud storage è più economico per archivio dati?

- A) Block Storage
- B) File Storage
- C) Object Storage
- D) Tutti uguali

Risposta: C

10. Quale tool è usato per deployare infrastruttura as code?

- A) Docker
- B) Terraform
- C) Kubernetes
- D) Jenkins

Risposta: B

Domande Aperte

1. Spiega la differenza tra un'API di Livello 0 e Livello 2 del Richardson Model, facendo un esempio pratico per ognuno.

Risposta suggerita:

- L.0 (Swamp of POX): POST /api/doGetUser con action nel body
- L.2 (HTTP Verbs): GET /api/users/123 con metodi HTTP appropriati
- Vantaggi L.2: semantica HTTP chiara, caching, simpler to understand

2. Descrivi il flow di una CORS preflight request, spiegando perché è necessaria.

Risposta suggerita:

- Browser invia OPTIONS prima di richiesta complessa
- Check se server permette cross-origin access
- Headers Access-Control-Allow-Methods, Access-Control-Allow-Headers
- Necessaria per sicurezza (no unsolicited requests da siti malevoli)

3. Confronta IaaS, PaaS, e SaaS in termini di responsabilità, vantaggi e svantaggi.

Risposta suggerita:

- Tabella di comparazione con chi gestisce cosa
- IaaS: max flessibilità, max responsabilità
- PaaS: buon equilibrio, rapido sviluppo
- SaaS: zero gestione, vendor lock-in massimo

4. Quali sono i vantaggi di un Cloud Storage Object (S3) rispetto a Block Storage, e quando useresti ognuno?

Risposta suggerita:

- Object: economico, scalabile, per non-structured data
- Block: performance, per database, transazioni
- S3 per: backup, archive, media
- Block per: database, applicazioni transazionali

5. Spiega il modello di Shared Responsibility nel Cloud e dai un esempio per ogni layer IaaS/PaaS/SaaS.

Risposta suggerita:

- Provider: infrastruttura sempre

- IaaS: utente gestisce SO, app, dati
 - PaaS: utente gestisce solo app, dati
 - SaaS: provider gestisce tutto
 - Esempio: Database MySQL (IaaS user gestisce, RDS PaaS provider gestisce)
-

CONCLUSIONE

Questo documento ti fornisce una base solida per preparare il tuo esame su API REST e Cloud Computing. I concetti sono spiegati in profondità con esempi pratici, diagrammi mentali, e casi d'uso reali.

Punti chiave da ricordare:

1. **REST non è SOAP:** REST è uno stile architetturale, non uno standard
2. **CRUD operations:** GET, POST, PUT/PATCH, DELETE mappano a Read, Create, Update, Delete
3. **Status codes importanti:** 200, 201, 400, 401, 404, 500 comunicano il risultato
4. **CORS è security:** Permette richieste cross-origin controllate
5. **Cloud modelli:** IaaS (infrastruttura), PaaS (piattaforma), SaaS (applicazione)
6. **Cloud deployment:** Public (economico), Private (sicuro), Hybrid (flessibile)
7. **Scalabilità cloud:** Elastica e on-demand

Studia questi argomenti in profondità e sarai preparato per l'esame!