

UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA



*Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione*

*Progettazione e Sviluppo di un sistema basato su tecnologie
serverless per l'interpolazione di dati geografici*

*Design and Development of a system based on serverless
technologies to interpolate geographical data*

Relatore:
DOTT. MANCINI ADRIANO

Laureando:
SGRIGNUOLI PIERO

ANNO ACCADEMICO 2018-2019

Indice

Elenco delle figure	4
1 Introduzione	5
1.1 Obiettivo	5
1.2 Struttura della Tesi	6
2 Cloud Computing e Amazon AWS	7
2.1 Cloud Computing	7
2.1.1 Tipologie di servizio	7
2.1.2 Approccio Serverless	9
2.2 Amazon AWS	10
2.2.1 AWS Lambda	10
2.2.2 Amazon API Gateway	11
2.2.3 Amazon Simple Storage Service	12
3 Strumenti e Metodi	15
3.1 Strumenti Utilizzati	15
3.1.1 Piattaforma AWS	15
3.1.2 Geographical Information System	23
3.2 Metodi di Interpolazione	25
3.2.1 Metodi non geostatistici	25
3.2.2 Metodi geostatistici	28
4 Sviluppo Applicazione Serverless	31
4.1 Progettazione	31
4.1.1 Analisi dei requisiti	31
4.1.2 Progettazione architetturale	32
4.2 Implementazione	33
4.2.1 LambdaInterpolator.py	33
4.2.2 LambdaGeoServer.py	36
4.3 Testing	39
4.3.1 Interpolazione dei dati	39
4.3.2 Pubblicazione su GeoServer	40

5 Conclusioni e Sviluppi Futuri	43
5.1 Conclusioni	43
5.2 Sviluppi futuri	44
Bibliografia	45

Capitolo 1

Introduzione

1.1 Obiettivo

Il progetto di tesi si concentra sulla progettazione e lo sviluppo di un'applicazione *serverless* per l'interpolazione di dati spaziali.

In un contesto di agricoltura di precisione, un sistema equipaggiato con Arduino, è programmato per rilevare la presenza di insetti in un'area. Mediante l'utilizzo di trappole adesive e algoritmi di *computer vision*, gli insetti intrappolati, vengono conteggiati grazie al sistema di visione ed il risultato viene trasmesso sul *cloud* per successive elaborazioni. L'applicazione, dunque, riceve in ingresso il set di dati contenente informazioni sulle coordinate geografiche delle trappole (latitudine e longitudine) e, per ogni trappola, il numero di insetti presenti. Successivamente, i dati sono interpolati da un apposito algoritmo e, terminata l'elaborazione, il risultato viene comunicato ad un server.

L'applicazione, nello specifico, consiste di due *Lambda Function* distribuite sul cloud Amazon Web Services (AWS). I dati da interpolare sono inviati tramite richiesta *HyperText Transfer Protocol over Secure Socket Layer* (HTTPS) all'applicazione web, la quale, terminata l'esecuzione, si interfaccia in maniera automatica con GeoServer, uno strumento *open source* per la visualizzazione e l'editing di dati geospaziali. GeoServer, infine, si occupa di applicare un gradiente di colore al risultato, producendo una mappa di calore (*heatmap*).

1.2 Struttura della Tesi

La parte essenziale del progetto di tesi è lo sviluppo di micro-servizi web con tecnologia *Serverless*, a cui segue l'interazione con GeoServer [9]. Inizialmente, sono affrontati i concetti alla base dell'approccio *Serverless* e del *Cloud Computing* in generale. Vengono poi introdotti tutti i servizi utilizzati, con accenni ai principali metodi di interpolazione. Segue, l'analisi in dettaglio dello sviluppo dell'applicazione e, infine, vengono esposti i possibili sviluppi futuri del progetto. La struttura dell'elaborato è dunque la seguente:

- nella prima parte vengono esposti i concetti dell'approccio *Serverless* e delle tecnologie di *Cloud Computing*, con un'introduzione ad Amazon AWS;
- nella seconda parte vengono introdotti gli strumenti ed i metodi utilizzati per lo sviluppo del progetto, con una breve descrizione dei principali algoritmi di interpolazione;
- nella terza parte si scende in dettaglio riguardo lo sviluppo dell'applicazione, analizzandone l'implementazione;
- nell'ultima parte vengono esposte le conclusioni e introdotti i possibili sviluppi futuri dell'applicazione.

Capitolo 2

Cloud Computing e Amazon AWS

2.1 Cloud Computing

Il *Cloud Computing*, nel suo significato più generale, indica la disponibilità "su richiesta" di risorse computazionali, per lo più potenza di calcolo e archiviazione dei dati. I primi riferimenti a questa tipologia di servizi risalgono al 1996 [17], ma è soltanto nell'agosto del 2006, con il lancio da parte di Amazon del servizio *Elastic Compute Cloud* (EC2) [6], che questa tecnologia diventa popolare e inizia ad attirare l'interesse dei professionisti.

Per l'utente, il vantaggio nell'utilizzo di tali servizi risiede principalmente nell'abbattimento dei costi di gestione, difatti, l'utilizzatore del servizio non deve preoccuparsi della manutenzione dell'infrastruttura, della protezione dei dati e dell'aggiornamento dei sistemi. Inoltre, non dovendo gestire un'infrastruttura fisica si hanno notevoli risparmi di tempo, e ciò permette di focalizzare tutti gli sforzi sullo sviluppo delle applicazioni, migliorando, indirettamente, la flessibilità di un'azienda nel rispondere alle variazioni di esigenze dei clienti, specialmente in contesti di *Agile Development*.

2.1.1 Tipologie di servizio

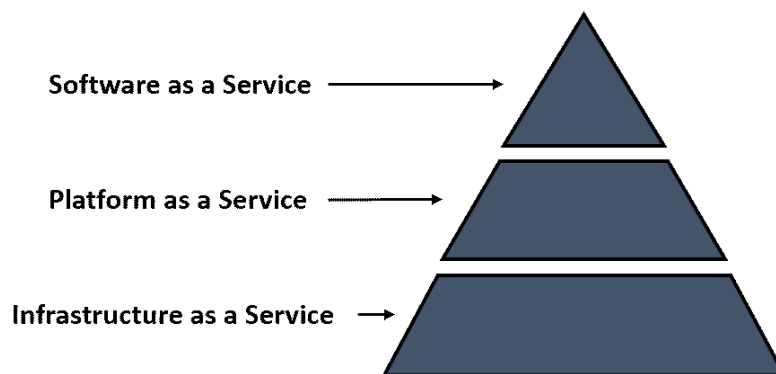
La tecnologia alla base del *Cloud Computing* è la virtualizzazione, cioè la separazione di singoli dispositivi fisici in più dispositivi "virtuali". Considerando che, nella maggior parte dei casi, un singolo utente non satura tutta la capacità computazionale di un singolo dispositivo fisico, la virtualizzazione permette quindi, di allocare le rimanenti risorse per lo svolgimento di altre attività, anche di utenti differenti. Si viene, dunque, a creare un sistema costituito da molteplici unità computazionali indipendenti, altamente scalabile e in grado di ripartire automaticamente le risorse, in modo da offrire a ogni utente le prestazioni di cui ha bisogno.

I provider di servizi *cloud* offrono principalmente tre modelli di servizio, ognuno con diverse caratteristiche di controllo, flessibilità e gestione. Si dividono in:

- **Software as a Service (SaaS)**: è un modello che racchiude applicativi e sistemi software, accessibili da un qualsiasi tipo di dispositivo (computer, smartphone, tablet) attraverso il semplice utilizzo di un'interfaccia *client*. In questo modo, l'utilizzatore non deve preoccuparsi di gestire le risorse e l'infrastruttura, in quanto controllati dal provider che li fornisce. Un chiaro esempio è la suite di prodotti Google.
- **Platform as a Service (PaaS)**: è un modello nel quale vengono situati i servizi di piattaforme online, grazie al quale un utente, di solito uno sviluppatore, può effettuare la distribuzione di applicazioni e servizi Web che intende fornire. In questo caso, l'utilizzatore può sviluppare ed eseguire le proprie applicazioni attraverso gli strumenti forniti dal provider, il quale garantisce il corretto funzionamento dell'infrastruttura sottostante. Alcuni esempi sono Amazon DynamoDB e Amazon API Gateway.
- **Infrastructure as a Service (IaaS)**: è un modello nel quale vengono messi a disposizione risorse hardware virtualizzate, affinché l'utilizzatore possa creare e gestire, secondo le esigenze, una propria infrastruttura sul *cloud*, senza preoccuparsi di dove siano allocate le risorse. Tra i numerosi esempi spiccano Amazon Elastic Cloud Compute (EC2), Amazon Simple Storage Service (S3), Amazon Virtual Private Cloud (VPC), Google Cloud Engine, Google Cloud Storage.

Le tipologie di servizi esposti rappresentano, inoltre, diversi livelli di astrazione, come schematizzato in figura 2.1.

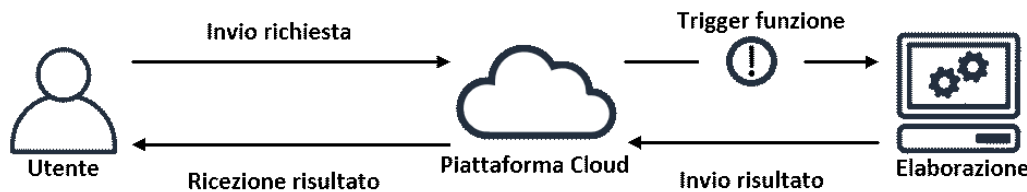
Figura 2.1: Livelli di astrazione del *Cloud Computing*



2.1.2 Approccio Serverless

Con *Serverless*, anche detto *Function as a Service* (FaaS), si identificano quei servizi *cloud* che permettono all'utilizzatore di sviluppare soluzioni senza occuparsi dell'implementazione dell'architettura server e dell'hardware su cui è ospitato.

Figura 2.2: Schema approccio serverless



Si espongono di seguito i principali vantaggi dell'approccio serverless:

- **Astrazione:** essere svincolati dal substrato hardware garantisce un elevato livello di astrazione. È possibile, infatti, sviluppare singole funzioni indipendenti tra loro, ognuna con un proprio *trigger* che ne scatena l'esecuzione. I *trigger* delle funzioni possono essere di svariato tipo, come l'esecuzione periodica oppure l'esecuzione in risposta ad un evento;
- **Scalabilità:** indipendentemente dalle richieste che l'applicazione riceve, le risorse vengono calibrate automaticamente, adattandosi alle necessità di memoria e *throughput*;
- **Costi:** la fatturazione per le applicazioni *serverless* segue spesso il modello "*Pay per Use*", anche detto "*Pay as You Go*", che permette di pagare solo il tempo effettivo di esecuzione del codice, evitando sprechi quando l'applicazione è inattiva.

Ovviamente, il *Serverless* presenta anche alcuni svantaggi, tra cui:

- **Vendor lock-in:** un'applicazione *serverless*, solitamente, nasce e muore in una piattaforma, raramente è possibile trasferire l'applicazione verso un altro provider nella sua interezza, vincolando l'utente a rimanere con un determinato gestore;
- **Inadeguato a lunghe attività:** l'approccio *serverless* è indicato per attività di breve durata, periodiche, come l'invio di una mail o un'elaborazione dati. Risulta inadeguato per attività continuative, dove le funzioni vengono eseguite costantemente. Infatti, in questi casi, il costo totale potrebbe superare quello di un'istanza di calcolo riservata;

- **Complessità:** la curva di apprendimento per le applicazioni *serverless* è spesso ripida. Il *testing* risulta difficoltoso e deve essere svolto su misura in base ai casi di studio, mentre il rilascio di aggiornamenti e la gestione delle versioni sono più complessi rispetto ad un'architettura monolitica.

Riepilogando, si nota come l'approccio *serverless* è sicuramente vantaggioso per via dei costi ridotti e della grande flessibilità che offre. Tuttavia, non può essere utilizzato per sviluppare qualsiasi tipologia di software, prediligendo le attività di breve durata e che non richiedono un elevato flusso di dati. Nella maggior parte dei casi, è utilizzato per applicazioni che ricevono una richiesta, effettuano un'elaborazione relativamente a tale richiesta e restituiscono il risultato.

2.2 Amazon AWS

Relativamente al progetto di tesi, si è scelto di utilizzare la piattaforma Amazon AWS [3]. *Amazon Web Services* (AWS) è una piattaforma *cloud* completa e all'avanguardia in grado di offrire numerosi servizi in ogni ambito del settore IT, dalle infrastrutture per il calcolo, l'archiviazione e i database, fino alle nuove tecnologie, quali *Machine Learning*, intelligenza artificiale e *Internet of Things* (IoT). Alla base di tutti questi servizi, è presente una vasta gamma di strumenti per la sicurezza, concetto chiave quando si parla di *Cloud Computing*.

AWS opera, inoltre, in molteplici regioni geografiche in tutto il mondo, garantendo una bassa latenza ed un throughput elevato pressoché ovunque.

In particolare, la scelta di questo provider, nel progetto svolto, è stata influenzata dall'ottimo piano gratuito offerto che ha permesso di sviluppare l'applicazione *serverless* rispettando appieno i requisiti richiesti.

Figura 2.3: Logo Amazon AWS



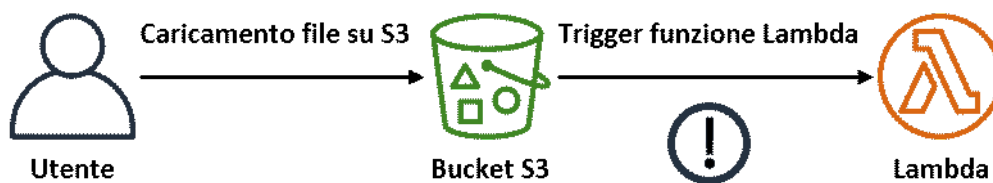
2.2.1 AWS Lambda

AWS Lambda [11] è un servizio di elaborazione *serverless* che esegue codice in risposta a determinati eventi, oltre a gestire le risorse di elaborazione autonomamente. Offre supporto nativo al linguaggio Java, Go, PowerShell, Node.js, C#, Python e Ruby. Solitamente, Lambda viene usato per estendere altri servizi AWS con logica personalizzata oppure per creare servizi di *back-end* in grado di sfruttare la scalabilità, le prestazioni e la sicurezza di AWS.

Per semplicità, con il termine “*funzione Lambda*” si identifica il codice della funzione con le relative informazioni di configurazione e requisiti in termini di risorse. Nello specifico, le funzioni in Lambda, hanno dei *trigger* in grado di avviarne l’esecuzione. Questi *trigger* possono essere di vario genere, tra cui richieste HTTP tramite Amazon API Gateway, modifiche a un bucket Amazon S3 o l’aggiornamento di tabelle in Amazon DynamoDB.

Le funzioni Lambda sono “*stateless*” cioè il loro ciclo di vita inizia con il *trigger* della funzione e finisce con il termine dell’esecuzione. Ciò permette ad AWS di lanciare in esecuzione più istanze di una funzione contemporaneamente, il tutto in maniera automatica, basandosi sulla frequenza delle richieste in entrata.

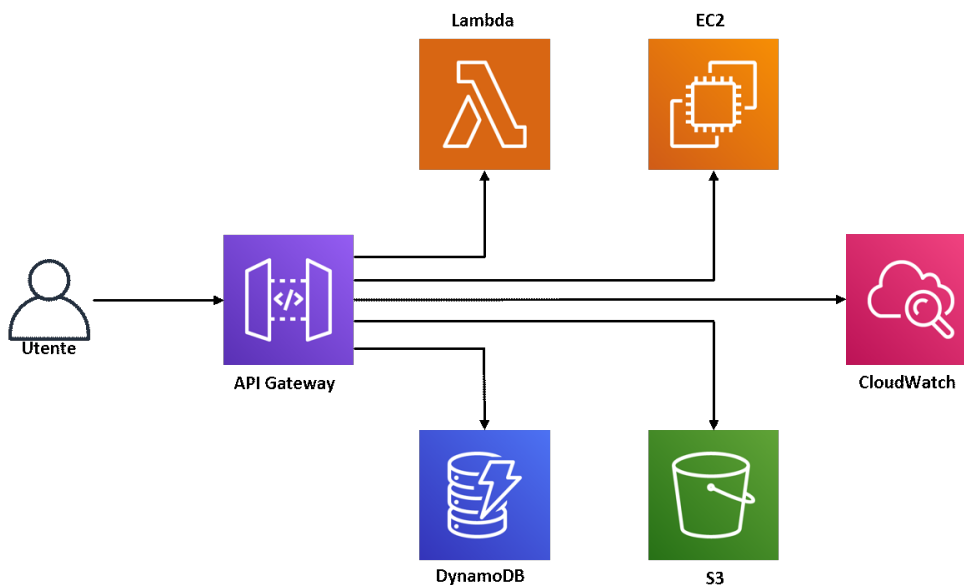
Figura 2.4: Esempio di trigger in Lambda



2.2.2 Amazon API Gateway

Amazon API Gateway [2] è un servizio completamente gestito che semplifica la creazione, la pubblicazione, la manutenzione, il monitoraggio e la protezione delle *Application Programming Interface* (API).

Figura 2.5: Interazioni di API Gateway



Con tale servizio, è possibile creare API *RESTful* utilizzando API HTTP o API *REST*. Le API HTTP sono ottimizzate per costruire API che eseguono il *proxy* sulle funzioni Lambda, rendendole ideali per carichi di lavoro *serverless*, mentre le API *REST* offrono la funzionalità *proxy* API e le funzionalità di gestione in un'unica soluzione. Inoltre, è possibile creare anche API *WebSocket*, che mantengono una connessione continua tra i *client* connessi per consentire la comunicazione in tempo reale.

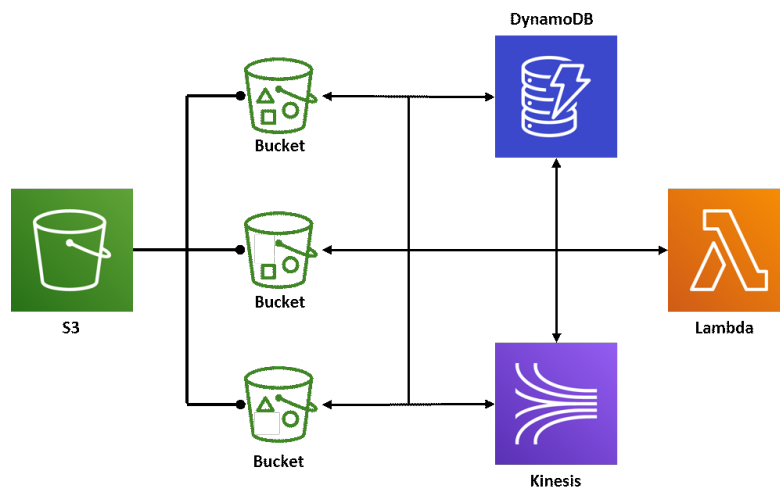
API Gateway è spesso utilizzato insieme alle funzioni Lambda dato che è possibile creare una API personalizzata per il codice in esecuzione su AWS Lambda, per poi effettuare l'invocazione alla funzione Lambda tramite API.

Sul fronte della sicurezza, per autorizzare e verificare le richieste API, è possibile sfruttare l'integrazione con AWS *Identity and Access Management* (IAM) e creare delle *policy* personalizzate per l'accesso alle API e agli altri servizi della piattaforma.

2.2.3 Amazon Simple Storage Service

Amazon *Simple Storage Service* (S3) [18] è un servizio per l'archiviazione di oggetti che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni all'avanguardia nel settore. Possiede, inoltre, varie caratteristiche utilizzabili per organizzare e gestire i dati in modi che supportano casi d'uso specifici.

Figura 2.6: Integrazione di S3 in altri servizi AWS



I dati sono archiviati come oggetti all'interno di "bucket", con funzionalità per aggiungere *tag* di metadati agli oggetti, spostare dati tra diversi *bucket*, configurare i controlli di accesso e proteggere i dati da utenti non autorizzati. S3, inoltre, supporta il controllo sulla versione dei dati, dispone di meccanismi per prevenire eliminazioni accidentali e rende possibile replicare i dati in diverse regioni AWS.

Come per gli altri servizi della piattaforma AWS, anche S3 presenta numerose funzionalità nel campo della sicurezza, supportando sia la crittografia lato server, con varie opzioni di gestione delle chiavi, sia la crittografia lato *client* per i caricamenti di dati.

Per impostazione predefinita, l'accesso ai *bucket* S3 non è pubblico, ma è riservato solo all'utente che ha creato tale *bucket*, tuttavia, grazie al servizio AWS IAM è possibile creare *policy* adatte ai singoli casi d'uso con facilità, come già visto per Amazon API Gateway. Inoltre, tramite le *Access Control List* (ACL) è possibile scendere nel dettaglio, definendo regole di accesso per i singoli oggetti in un *bucket*.

Capitolo 3

Strumenti e Metodi

3.1 Strumenti Utilizzati

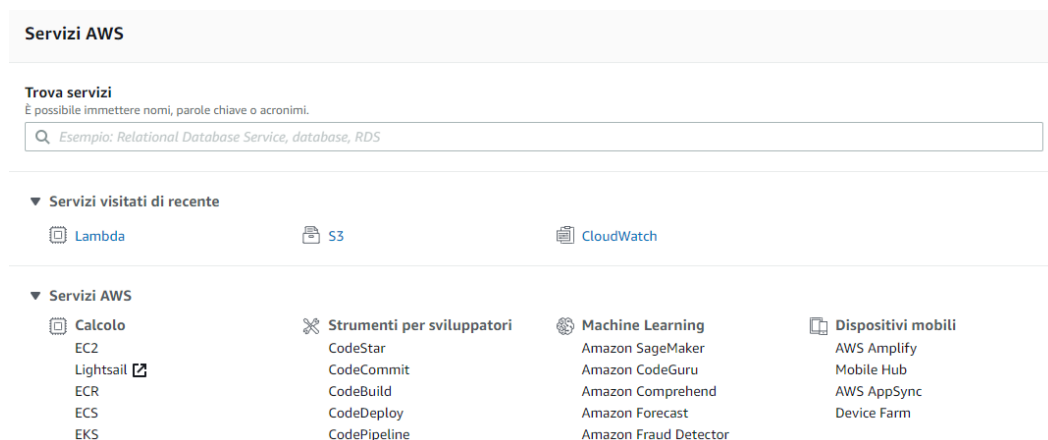
Come già accennato nel capitolo 2, per lo sviluppo dell'applicazione *serverless* è stata usata la piattaforma cloud AWS, in aggiunta ad altri strumenti complementari come QGIS [16] e GeoServer [9].

Nei seguenti paragrafi vengono esposti, dunque, tutti gli strumenti e i servizi utilizzati.

3.1.1 Piattaforma AWS

La piattaforma cloud AWS è composta da numerosi servizi suddivisi in varie categorie. Dopo l'accesso con le proprie credenziali, viene mostrata la console di gestione (Figura 3.1), che permette di selezionare velocemente il servizio di cui si ha bisogno. In particolare, per il progetto di tesi sono stati utilizzati: AWS IAM, Amazon S3, AWS Lambda, Amazon API Gateway e Amazon CloudWatch.

Figura 3.1: Console di gestione AWS



3.1.1.1 AWS Identity and Access Management

AWS Identity and Access Management (IAM) consente di gestire in sicurezza l'accesso ai servizi e alle risorse AWS. Grazie a IAM, è possibile creare e gestire gruppi di utenti per consentire o negare l'accesso alle varie risorse della piattaforma.

Figura 3.2: Pannello di riepilogo IAM

The screenshot shows the IAM console interface for a user. At the top, it displays the user's ARN, path, and creation date. Below this, there are tabs for 'Autorizzazioni', 'Gruppi', 'Tag', 'Credenziali di sicurezza', and 'Consulente accessi'. The 'Autorizzazioni' tab is selected, showing a list of applied policies. The policies listed are:

Nome policy	Tipo di policy
Collegate direttamente	
AWSLambdaFullAccess	Policy gestita da AWS
IAMFullAccess	Policy gestita da AWS
AmazonAPIGatewayInvokeFullAccess	Policy gestita da AWS
AmazonAPIGatewayAdministrator	Policy gestita da AWS

In figura 3.2 è mostrato il pannello di riepilogo di IAM, in particolare, si notano i criteri di sicurezza assegnati all'utente "SharedUser". Nel progetto di tesi in questione, non vi erano particolari vincoli sulle policy di sicurezza, all'utente, dunque, sono stati assegnati direttamente i poteri necessari all'invocazione delle funzioni Lambda e per l'utilizzo di API Gateway.

Figura 3.3: Chiave d'accesso IAM

ID chiave di accesso	Data creazione	Ultimo utilizzo	Stato
AKIAS[REDACTED]75GR	2019-06-09 21:34 UTC+0100	2019-12-21 21:41 UTC+0...	Attivo Rendi inattivo ✕

La Figura 3.3 presenta un ulteriore pannello di IAM che mostra le chiavi di accesso necessarie per rendere sicure le richieste del protocollo REST o HTTP alle API del servizio AWS. Queste chiavi d'accesso sono necessarie, nello specifico del progetto, per interfacciarsi con il servizio API Gateway.

3.1.1.2 S3

Il servizio Amazon S3 è stato utilizzato per la memorizzazione del risultato dell'interpolazione. L'utilizzo di questo strumento è molto semplice e immediato, basta infatti creare un nuovo *bucket* e scegliere una classe di *storage* tra quelle disponibili e si può essere già operativi.

Nel progetto in esame non erano presenti necessità specifiche, è stata scelta quindi, la classe di *storage* standard. La figura 3.4 mostra la pagina di riepilogo del *bucket*.

Figura 3.4: Riepilogo bucket S3

The screenshot shows the AWS S3 console interface for a bucket named 'awsapp-storage'. At the top, there are navigation tabs: 'Panoramica' (selected), 'Proprietà', 'Autorizzazioni', 'Gestione', and 'Punti di accesso'. Below the tabs is a search bar with the placeholder text 'Digita un prefisso e premi Invio per avviare la ricerca. Premi ESC per cancellare.' Below the search bar are action buttons: 'Carica', 'Crea cartella', 'Scarica', and 'Operazioni'. The region is set to 'UE (Francoforte)'. The main content area shows a table with columns for 'Nome', 'Ultima modifica', 'Dimensioni', and 'Classe di storage'. The table contains two entries: 'img' and 'tmp', both with dashes in the 'Ultima modifica' and 'Dimensioni' columns. The table is labeled 'Visualizzazione 1 in 2'.

Cliccando su una cartella appartenente al *bucket* si ottiene l'eventuale lista di file (figura 3.5).

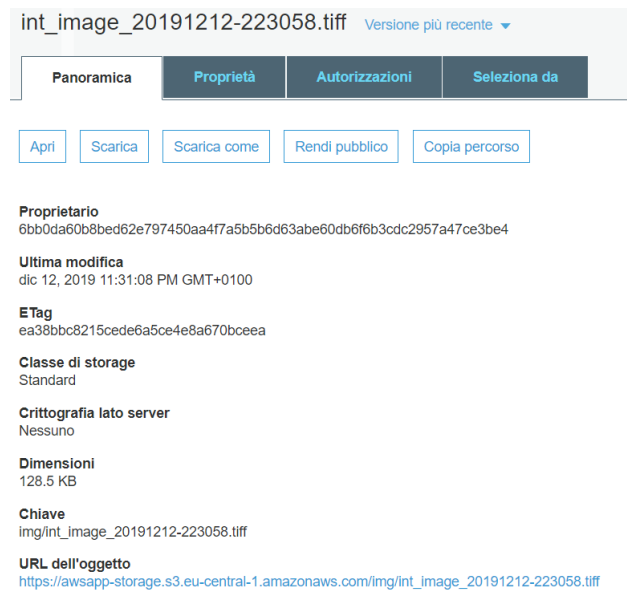
Figura 3.5: Cartella S3

The screenshot shows the AWS S3 console interface for a folder within the 'awsapp-storage' bucket. The 'Panoramica' tab is selected. Below the search bar are action buttons: 'Carica', 'Crea cartella', 'Scarica', and 'Operazioni'. The region is set to 'UE (Francoforte)'. The main content area shows a table with columns for 'Nome', 'Ultima modifica', 'Dimensioni', and 'Classe di storage'. The table contains four entries, all with a file icon and a name starting with 'int_image_'. The 'Ultima modifica' column shows dates and times, and the 'Dimensioni' column shows '128.5 KB'. The 'Classe di storage' column shows 'Standard'. The table is labeled 'Visualizzazione 1 in 4'.

Selezionando invece un file, si apre la pagina che ne riepiloga le proprietà principali, tra cui, il proprietario dell'oggetto, la data dell'ultima modifica, le dimensioni del file e il suo URL.

Come evidenziato nelle figure 3.4, 3.5 e 3.6 l'interfaccia di S3 è molto semplice e minimale, essendo il servizio pensato per la memorizzazione di file a breve e medio termine. Solitamente, tale servizio, viene integrato con altre risorse AWS, nel progetto di tesi viene utilizzato come supporto alle funzioni *Lambda*.

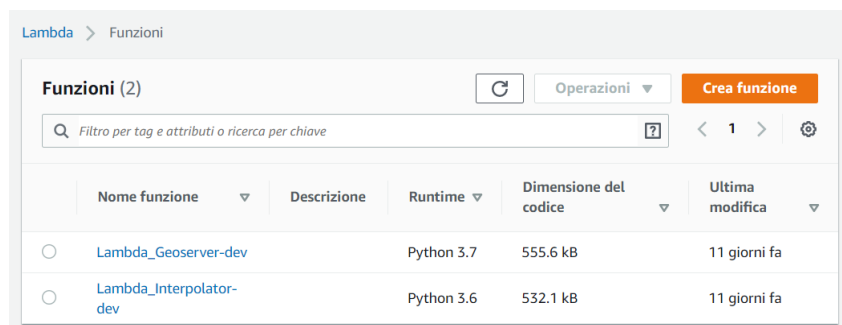
Figura 3.6: Proprietà di un file su S3



3.1.1.3 Lambda

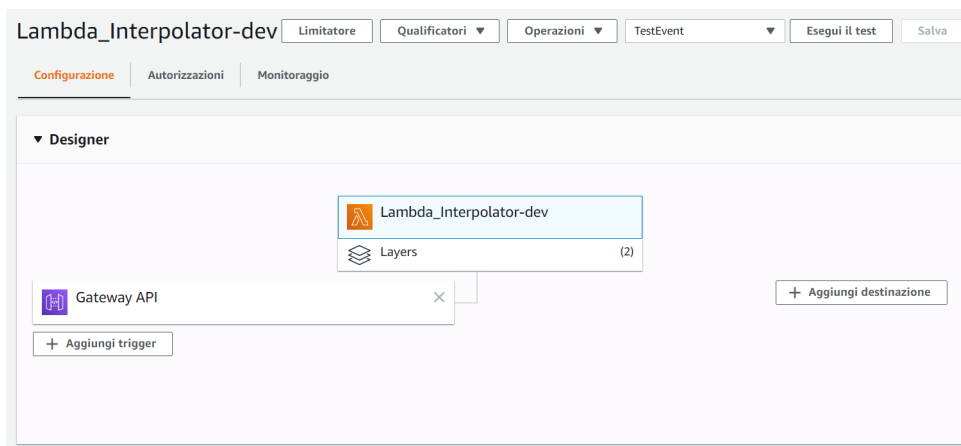
AWS *Lambda* rappresenta lo strumento chiave del progetto, ospita, infatti, il codice necessario per l'interpolazione dei dati e per la comunicazione con GeoServer. Un totale di due funzioni Lambda sono state create per l'applicazione *serverless*, dove ognuna di esse svolge un compito specifico. La figura 3.7 mostra la lista di funzioni Lambda create.

Figura 3.7: Lista funzioni Lambda



Una volta selezionata la funzione di interesse, il servizio mostra la schermata presentata in figura 3.8 che, oltre a varie opzioni di configurazione, contiene il pannello “designer”. In questo pannello è possibile impostare facilmente i *trigger* della funzione Lambda ed eventuali destinazioni.

Figura 3.8: Designer Lambda



In figura 3.8 si può notare la scheda “Layers”. I *Layer* sono una funzionalità molto utile di AWS Lambda, difatti, se il codice dipende da moduli esterni, rispetto a quelli nativi del linguaggio scelto, è possibile caricarli separatamente sotto forma di *Layer*, mantenendo così, la dimensione della funzione Lambda molto piccola, agevolando la distribuzione di nuove versioni. Ogni funzione può avere fino a cinque *Layer* e non è necessaria alcuna impostazione aggiuntiva dato che Lambda include automaticamente i moduli caricati nei *Layer* durante l’esecuzione del codice.

L’uso di questa funzionalità non è obbligatoria, se il codice necessita di moduli di piccole dimensioni, il pacchetto contenente il codice della funzione e gli eventuali moduli aggiuntivi, può essere caricato direttamente su AWS Lambda oppure attraverso S3 se il pacchetto è di dimensioni eccessive. Tuttavia, una funzione Lambda, inclusi i relativi *Layer* non può superare i 250 MB.

Nel caso di funzioni che non superano i 3 MB, è possibile modificare o scrivere codice direttamente nell’editor di testo messo a disposizione da AWS Lambda, come mostrato in figura 3.9. Considerando che, normalmente, il testing delle funzioni non viene eseguito direttamente su Lambda, si hanno a disposizione le variabili d’ambiente (figura 3.10). Esse risultano molto comode per impostare alcune configurazioni relative all’ambiente (*environment* o *env*) di sviluppo in cui ci si trova, permettendo di lavorare sia su AWS Lambda che in locale senza aggiornare ogni volta il codice.

Figura 3.9: Console per la modifica del codice

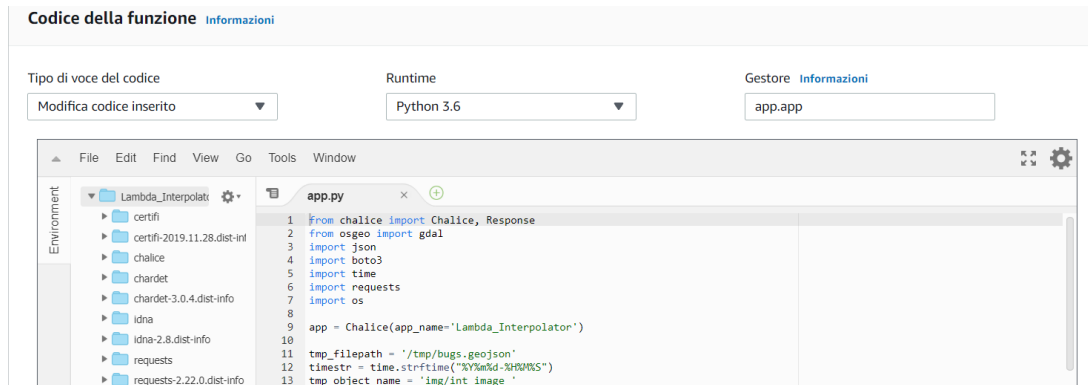
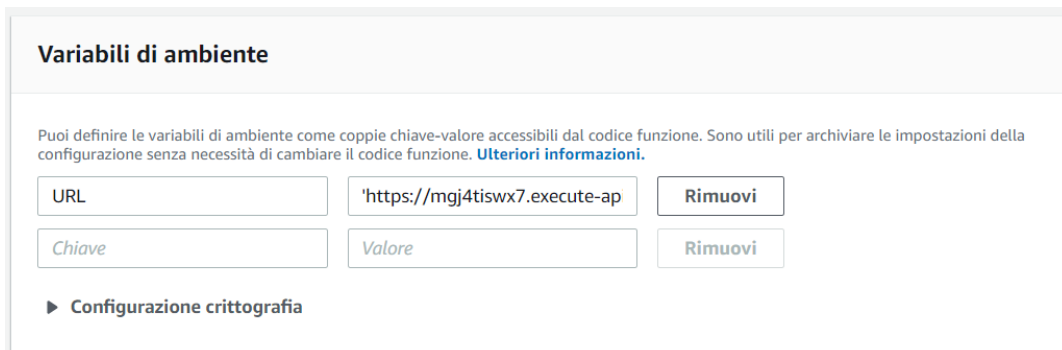


Figura 3.10: Configurazione variabili d'ambiente su Lambda



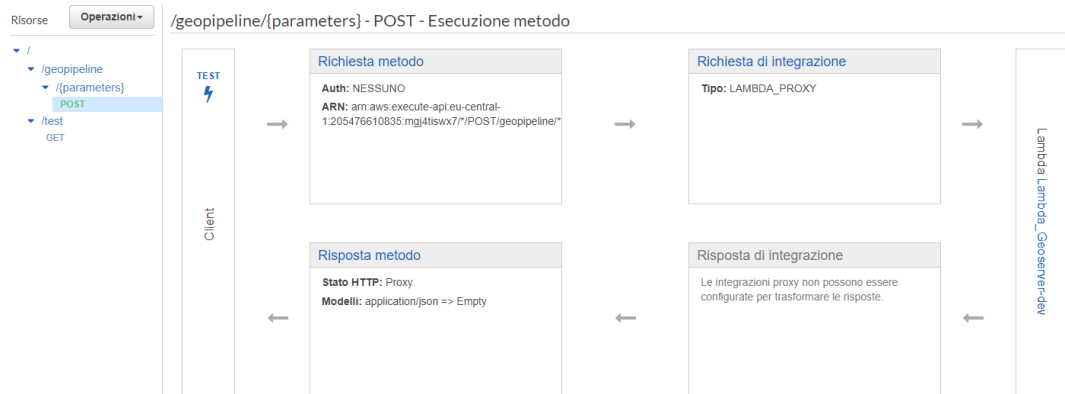
3.1.1.4 API Gateway

Nello sviluppo dell'applicazione *serverless*, il servizio Amazon API Gateway è stato utilizzato come *trigger* delle funzioni Lambda a seguito di richieste HTTP, usando il protocollo *REST*. Selezionando l'API di interesse, viene mostrato il suo elenco di risorse e i relativi metodi HTTP associati. Sulla destra si ha il pannello di test, che mostra in anteprima le modifiche alla configurazione e permette di testare l'API selezionata direttamente nella console API Gateway.

La figura 3.11 riporta il pannello di configurazione relativo al metodo *POST* della risorsa “{parameters}”.

Sempre in riferimento alla figura 3.11, si nota la sezione “Richiesta metodo” che permette di configurare l'interfaccia pubblica dell'API, cioè la definizione dell'API proposta agli utenti. Questa definizione comprende l'autorizzazione e la definizione dei verbi HTTP che consentono un corpo (*payload*) in entrata, le intestazioni e i parametri di stringa di una richiesta.

Figura 3.11: Pagina di configurazione di API Gateway



La sezione “Richiesta di integrazione” permette di specificare come l’API comunica con il *back-end* (Lambda, HTTP o altri servizi AWS) e come i dati della richiesta devono essere trasformati prima di essere inviati al back-end del metodo. Ad esempio, le funzioni Lambda non possono ricevere intestazioni (*headers*), interrogazioni (*query*) o parametri di stringa, ma è possibile utilizzare API Gateway per creare un evento *JSON* che contiene tutti i valori della richiesta.

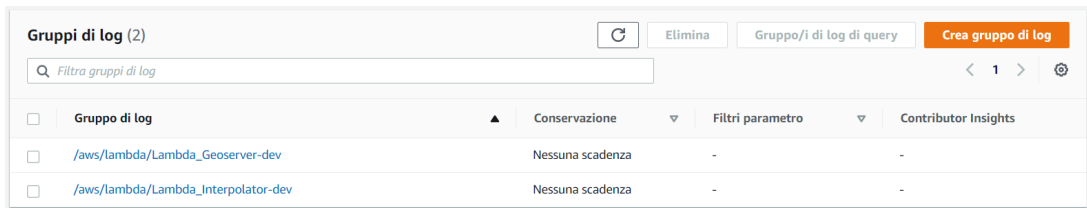
Infine, analogamente alla sezione “Richiesta metodo”, nella sezione “Risposta metodo” si possono specificare quali codici di stato HTTP sono supportati dal metodo in questione e, per ogni codice di stato, quali intestazioni può restituire.

Un’altra funzionalità molto utile sono le chiavi API, esse consentono di creare e distribuire delle chiavi di accesso per consentire a sviluppatori di terze parti di accedere alle API. Tuttavia, nel progetto in questione questa funzionalità non era richiesta.

3.1.1.5 CloudWatch

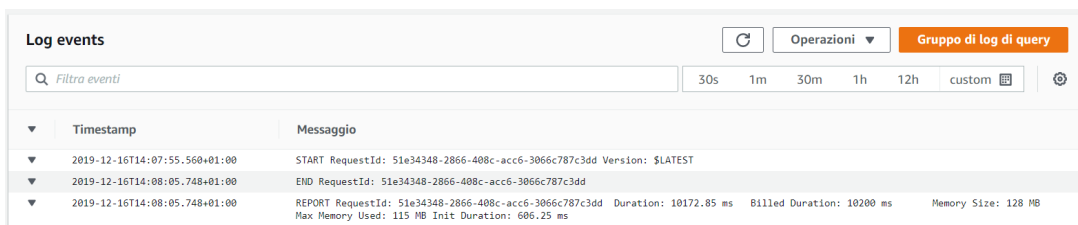
Amazon CloudWatch è un servizio di monitoraggio che fornisce dati e analisi concrete per monitorare le applicazioni, rispondere ai cambiamenti di prestazioni a livello di sistema e ottimizzare l’utilizzo delle risorse. CloudWatch raccoglie dati sotto forma di log, parametri ed eventi, fornendo una visualizzazione unificata delle risorse, delle applicazioni e dei servizi in esecuzione su AWS. Si può utilizzare, inoltre, per rilevare comportamenti anomali negli ambienti AWS, impostare allarmi e intraprendere azioni automatiche al fine di ripristinare situazioni ottimali. Come si vede in figura 3.12, CloudWatch raggruppa automaticamente i *log* in gruppi relativi alla stessa entità. In figura 3.13 è possibile notare, invece, il dettaglio di un *log*, in questo caso di una funzione Lambda. Nel *log* appaiono alcune informazioni sull’utilizzo effettivo di memoria e il tempo di esecuzione addebitato.

Figura 3.12: Lista gruppi di log su CloudWatch



Nel caso di errori nel codice della funzione, tra gli eventi del *log*, vengono mostrati anche i relativi codici di errore lanciati dal terminale.

Figura 3.13: Dettaglio di un log



3.1.1.6 Testing locale con Chalice

Eseguire il *testing* delle funzioni Lambda direttamente sulla piattaforma AWS è un approccio adatto solo a piccole porzioni di codice, ma per le applicazioni più complesse, risulta decisamente scomodo e difficoltoso. Difatti, per vedere il risultato dell'esecuzione di una funzione Lambda, bisogna andare nel servizio CloudWatch e analizzare i *log* relativi all'ultima chiamata della funzione associata, per poi procedere all'eventuale correzione. Tutto ciò porta via un quantitativo di tempo non indifferente.

Un'altra problematica sorge nei casi in cui le funzioni Lambda devono scrivere o leggere informazioni dal disco, infatti, AWS non permette alle funzioni mandate in esecuzione di scrivere sul *File System*, a parte alcune eccezioni. Risulta, quindi, più naturale e semplice testare il codice in un ambiente di sviluppo locale e, quando si è sicuri del risultato, caricare il codice su AWS.

Per un *testing* più agile si è scelto di utilizzare Chalice [5], un *framework* per scrivere applicazioni *serverless* in linguaggio Python, pensato appositamente per AWS Lambda. Chalice fornisce uno strumento da riga di comando per creare, distribuire e gestire le applicazioni. Gestisce, inoltre, l'integrazione con API Gateway, S3, IAM e altri servizi AWS in modo automatico.

In figura 3.14 è riportato un esempio di distribuzione di una funzione su AWS Lambda con Chalice.

Figura 3.14: Esempio di deploy con Chalice

```
(AWS-Project) C:\Users\deeps\Desktop\AWS-Project\test-project>chalice deploy
Creating deployment package.
Creating IAM role: test-project-dev
Creating lambda function: test-project-dev
Creating Rest API
Resources deployed:
- Lambda ARN: arn:aws:lambda:eu-central-1:205476610835:function:test-project-dev
- Rest API URL: https://ekif8w7zm7.execute-api.eu-central-1.amazonaws.com/api/
```

Dopo aver creato un progetto in Chalice, con il comando “chalice local” viene lanciato un server HTTP locale che può essere utilizzato per il *testing*. In figura 3.15, nello specifico, viene fatta una richiesta “GET” al server che, in risposta, fornisce un semplice messaggio e il codice di stato della richiesta. Viene effettuata, poi, una richiesta “POST” a cui il server non è in grado di rispondere, quindi, informa l’utente che il metodo non è supportato e fornisce il relativo codice di errore.

Figura 3.15: Testing in locale con Chalice

```
(AWS-Project) C:\Users\deeps\Desktop\AWS-Project\test-project>chalice local
Serving on http://127.0.0.1:8000
This message is from the Python console
127.0.0.1 - - [09/Feb/2020 15:13:34] "GET / HTTP/1.1" 200 -
Serving on http://127.0.0.1:8000
Error 405 Method not allowed, use GET
127.0.0.1 - - [09/Feb/2020 15:24:19] "POST / HTTP/1.1" 405 -
```

3.1.2 Geographical Information System

Un *Geographical Information System*, generalmente abbreviato in GIS, è un sistema pensato per acquisire, registrare, analizzare e visualizzare dati geospaziali. GIS è utilizzato sia per processi di *problem solving* sia per *decision making*, grazie alla combinazione di strumenti geografici con strumenti di analisi statistica.

Nei sistemi GIS i dati sono rappresentati principalmente con due modelli: vettori e *raster*. I vettori sono usati per rappresentare punti, linee, poligoni o tutti quei dati che hanno una precisa posizione. I *raster*, invece, vengono usati per rappresentare superfici, temperature, precipitazioni e caratteristiche del terreno.

3.1.2.1 QGIS

QGIS è un’applicazione GIS *open source*, utilizzata per analizzare o generare dati spaziali. Nell’ambito del progetto di tesi è stata utilizzata per generare una collezione di dati, al fine di testare le funzionalità dell’applicazione *serverless*.

In Figura 3.16 si riporta un esempio dell'utilizzo di QGIS per generare dati relativi alla distribuzione di insetti in un'area.

Figura 3.16: Generazione di dati spaziali con QGIS

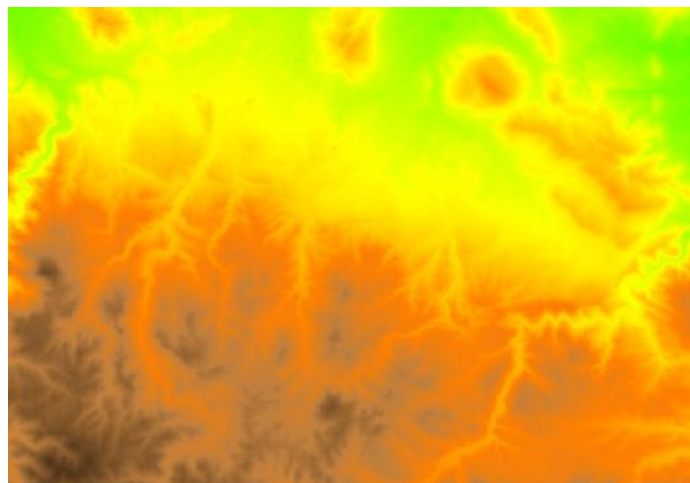


3.1.2.2 GeoServer

GeoServer è un *server open source* che permette agli utenti di visualizzare, modificare e condividere dati geospaziali utilizzando standard “aperti”.

Nel progetto è stato utilizzato per pubblicare il risultato dell'interpolazione dei dati in maniera automatica, interfacciando AWS Lambda con le *REST API* di GeoServer. La figura 3.17 mostra un esempio di immagine interpolata pubblicata su GeoServer.

Figura 3.17: Esempio di Layer pubblicato su GeoServer



3.2 Metodi di Interpolazione

L'interpolazione spaziale è il processo con il quale a partire da dei punti nello spazio (latitudine, longitudine ed altitudine) ai quali sono associati dei valori (es. temperatura, umidità) si vuole stimare i valori di altri punti sconosciuti (es. mediante una griglia regolare). Trova applicazioni in molti contesti, tra cui la geografia, geologia, ecologia, meteorologia, agronomia e agricoltura di precisione. Negli anni sono stati sviluppati numerosi metodi di interpolazione spaziale in diversi contesti. Nei sottoparagrafi 3.2.1 e 3.2.2, vengono esposti alcuni dei metodi più utilizzati in ambito GIS.

3.2.1 Metodi non geostatistici

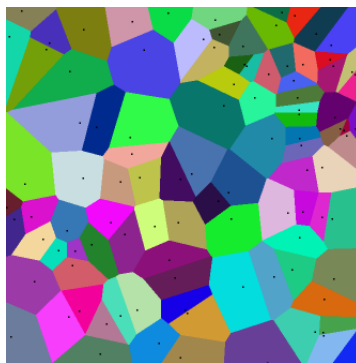
Di seguito, sono esposti tre metodi non geostatici, spesso utilizzati grazie alla loro bassa complessità concettuale.

3.2.1.1 Vicino più prossimo

Il metodo “Vicino più prossimo” [13] calcola il valore di un attributo in un punto non campionato basandosi sul valore dei campioni più vicini.

Nello specifico vengono generate delle bisettrici perpendicolari tra i punti campionati, formando dei poligoni di Voronoi. A ogni poligono corrisponde un campione e tutti i punti all'interno dello stesso poligono assumono il medesimo valore.

Figura 3.18: Esempio di poligoni di Voronoi



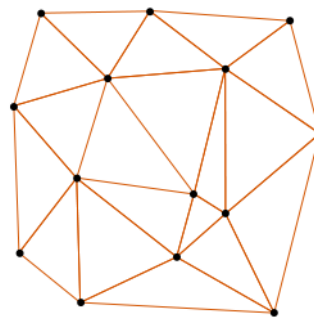
I vantaggi di questo metodo sono certamente la sua semplicità concettuale e, eventualmente, di implementazione. Tuttavia, non fornisce alcuna indicazione sull'errore di stima e assume implicitamente che la distribuzione sia omogenea e isotropa. Inoltre, il risultato “visivo” è spesso mediocre.

3.2.1.2 Rete irregolare di triangoli

Una rete irregolare di triangoli (TIN) [19] è la rappresentazione di una superficie continua costituita interamente da facce triangolari, usata principalmente nel campo del *Digital Elevation Modelling* (DEM). Una TIN (Figura 3.19) consiste in una rete triangolare di vertici, ognuno con le proprie coordinate tridimensionali, collegati tra loro in modo da formare una tassellazione triangolare. Il suo campo di applicazione più comune è la generazione di modelli digitali di superfici, calcolati sulla base dei dati di elevazione della superficie stessa.

Figura 3.19: Esempio di rete irregolare di triangoli

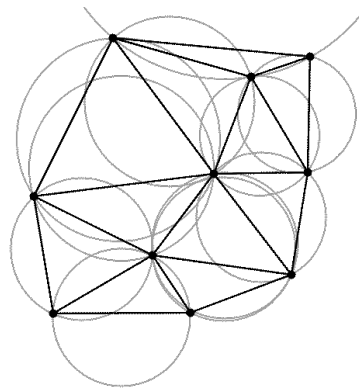
Fonte: https://en.wikipedia.org/wiki/Triangulated_irregular_network



La triangolazione viene effettuata, solitamente, con il metodo di Delaunay (figura 3.20), mentre il valore di un punto sconosciuto all'interno di un triangolo è stimato con un'interpolazione polinomiale lineare o cubica.

Figura 3.20: Triangolazione di Delaunay

Fonte: https://en.wikipedia.org/wiki/Delaunay_triangulation



Questo metodo non è indicato in tutti i casi. Un vantaggio è che i punti sono distribuiti in modo variabile, sulla base di un algoritmo che determina quali punti siano più significativi per rappresentare la superficie in maniera accurata. D'altro canto, se i dati in ingresso non sono ottimali, possono venire fuori molti spigoli seghettati nel risultato, distorcendo l'immagine.

3.2.1.3 Distanza inversa ponderata

Il metodo “Distanza inversa ponderata” (IDW) è un metodo che stima il valore di un attributo, nei punti non campionati, usando una combinazione lineare dei valori nei punti conosciuti, pesati secondo una funzione inversa della distanza del punto di interesse dai punti campionati.

L’assunzione di base è che i punti campionati, più vicini a quelli sconosciuti, sono più simili a quest’ultimi di quanto lo siano quelli che si trovano a distanze maggiori. Il peso dei singoli punti sulla media può essere espresso come:

$$\lambda_i = \frac{1/d_i^p}{\sum_{i=1}^n 1/d_i^p} \quad (3.1)$$

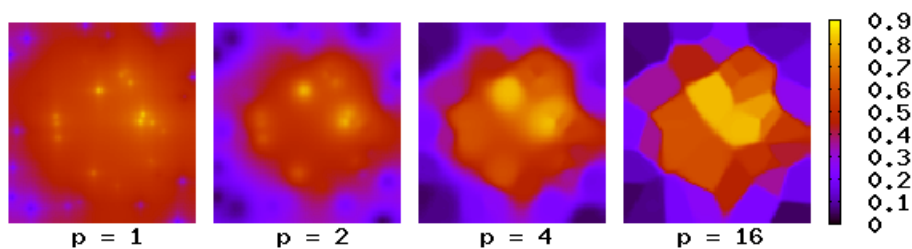
dove d_i è la distanza tra un punto x_0 e un punto x_i e n rappresenta il numero di punti campione usati per la stima.

Il fattore principale che influenza l’accuratezza del metodo IDW è l’esponente p . Il peso che hanno i campioni sulla media diminuisce all’aumentare della distanza, così i campioni più vicini hanno un peso maggiore sulla stima. Ciò risulta in un’interpolazione di tipo locale, ovvero vengono presi in considerazione, per la stima, solo i punti entro una certa area.

La scelta del valore dell’esponente p è arbitraria, così come l’area di ricerca dei punti, ma una scelta molto popolare è $p = 2$ e prende il nome di “Distanza inversa quadrata” (IDS) [12]. In figura 3.21, un esempio di IDW con diversi valori di p .

Figura 3.21: Esempio di interpolazione con IDW al variare di p

Fonte: https://en.wikipedia.org/wiki/Inverse_distance_weighting



Un vantaggio di questo metodo è la velocità di calcolo, difatti, essendo il raggio di ricerca e il valore dell’esponente arbitrari, è facile tenere sotto controllo la complessità computazionale. Tuttavia, il risultato è variabile, dato che dipende dalla funzione usata per assegnare i pesi ai campioni e dall’area di ricerca. Inoltre, il metodo, è sensibile alla presenza di molti punti vicini tra loro, generando in tal caso, un risultato non ottimale.

3.2.2 Metodi geostatistici

La geostatistica è la branca della statistica che si occupa dell'analisi di dati geografici. Si crede derivi dagli studi di Krige (1951) [10] in geologia mineraria, motivo per cui, i metodi in questione prendono il nome di “Kriging”.

I metodi di Kriging sono molto numerosi, ognuno ha le proprie caratteristiche e campi di applicazione. Nei sottoparagrafi 3.2.2.3 e 3.2.2.4 vengono esposti due metodi che forniscono le basi concettuali per tutti gli altri.

3.2.2.1 Semivarianza e variogramma

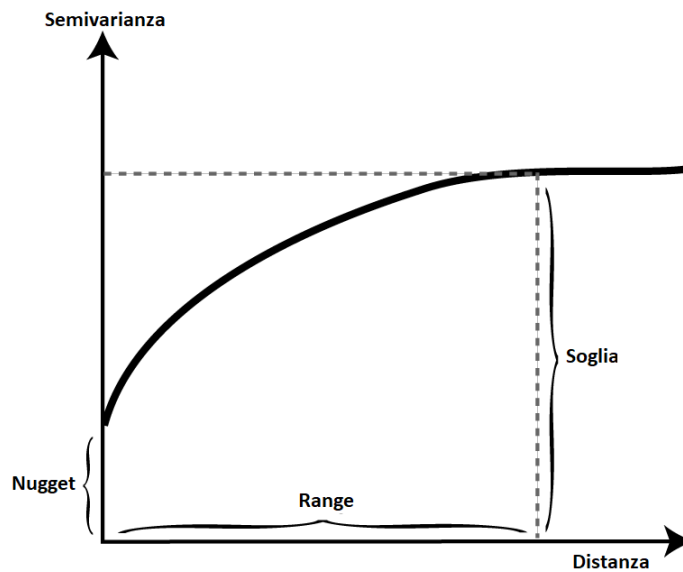
La semivarianza γ di Z tra due punti è un concetto fondamentale della geostatistica [12], è definita come:

$$\gamma(x_i, x_0) = \gamma(h) = \frac{1}{2} \text{var}[Z(x_i) - Z(x_0)] \quad (3.2)$$

dove h è la distanza tra il punto x_i e x_0 . $\gamma(h)$ è chiamato variogramma [21].

Il grafico di $\gamma(h)$ rispetto a h è chiamato semivariogramma, ne è riportato un esempio in figura 3.22.

Figura 3.22: Esempio di semivariogramma



Il grafico in figura 3.22, inoltre, mostra alcune caratteristiche importanti. Il “nugget” è un valore positivo di $\gamma(h)$ per h vicino a 0 (ricordando che h è la distanza) e rappresenta il livello di variabilità casuale. Il “range” rappresenta il valore della distanza per il quale viene raggiunta la “soglia”. La soglia, infine, rappresenta il valore per il quale la semivarianza si assesta.

I campioni separati da una distanza maggiore del range sono considerati spazialmente indipendenti, il range, inoltre, fornisce informazioni sulla grandezza della finestra di ricerca usata nei metodi di interpolazione spaziale [12].

3.2.2.2 Stimatore di Kriging

Tutti i metodi di Kriging, per calcolare il peso dei campioni sulla stima, si basano sulla seguente equazione:

$$\hat{Z}(x_0) - \mu = \sum_{i=1}^n \lambda_i [Z(x_i) - \mu(x_0)] \quad (3.3)$$

dove μ è una media stazionaria nota, assunta costante su tutto il dominio e calcolata come valore medio dei punti. Il parametro λ_i è il peso di Kriging, n è il numero di punti campionati usati per fare la stima e dipende dalla finestra di ricerca, infine, $\mu(x_0)$ è la media dei campioni all'interno della finestra di ricerca [20]. I pesi di Kriging sono calcolati minimizzando la varianza.

3.2.2.3 Kriging semplice

Il Kriging semplice è un metodo monovariabile la cui stima si basa sul seguente modello:

$$\hat{Z}(x_0) - \mu = \sum_{i=1}^n \lambda_i Z(x_i) + [1 - \sum_{i=1}^n \lambda_i] \mu. \quad (3.4)$$

Questo metodo è usato per stimare i residui dal valore di riferimento μ dato a priori, motivo per cui, viene anche chiamato “Kriging con media nota”.

Dato che il Kriging semplice è un metodo “*unbiased*”, il termine $1 - \sum_{i=1}^n \lambda_i$ non è necessariamente 0, e maggiore è il suo valore, più la stima del valore converge verso la media, e in generale il valore di $1 - \sum_{i=1}^n \lambda_i$ aumenta nei punti con meno campioni [1].

3.2.2.4 Kriging ordinario

Il Kriging ordinario è un altro metodo monovariabile, molto simile al Kriging semplice. Le uniche differenze sono la sostituzione del termine μ con una media locale $\mu(x_0)$ dei campioni nell'area di ricerca e l'imposizione di $[1 - \sum_{i=1}^n \lambda_i] = 0$ da

cui segue $\sum_{i=1}^n \lambda_i = 1$. Sostanzialmente, il Kriging ordinario, stima la media locale costante e successivamente applica il Kriging semplice.

Capitolo 4

Sviluppo Applicazione Serverless

4.1 Progettazione

Nel presente paragrafo sono esposti i requisiti progettuali al fine di formalizzare le funzionalità richieste e gli obiettivi da raggiungere nello sviluppo dell'applicazione *serverless*. Dopo aver discusso i requisiti funzionali, sono descritti i processi che hanno portato a determinate scelte di progettazione.

4.1.1 Analisi dei requisiti

Prima di descrivere i requisiti, è necessario esporre il problema di partenza. L'obiettivo da raggiungere è automatizzare un processo di interpolazione dati e pubblicazione del relativo risultato. A tale scopo, l'applicazione che si vuole creare, necessita inizialmente di una funzione che permetta di ricevere dei dati tramite una richiesta HTTP. Successivamente, un algoritmo di interpolazione deve elaborare questi dati e restituire il risultato. Infine, un'ulteriore funzione, ha il compito di generare una richiesta HTTP verso un *server*. Tale richiesta deve contenere tutti i parametri necessari alla pubblicazione del risultato ottenuto. In figura 4.1 è mostrata una schematizzazione grafica dei requisiti funzionali.

Figura 4.1: Schematizzazione dei requisiti

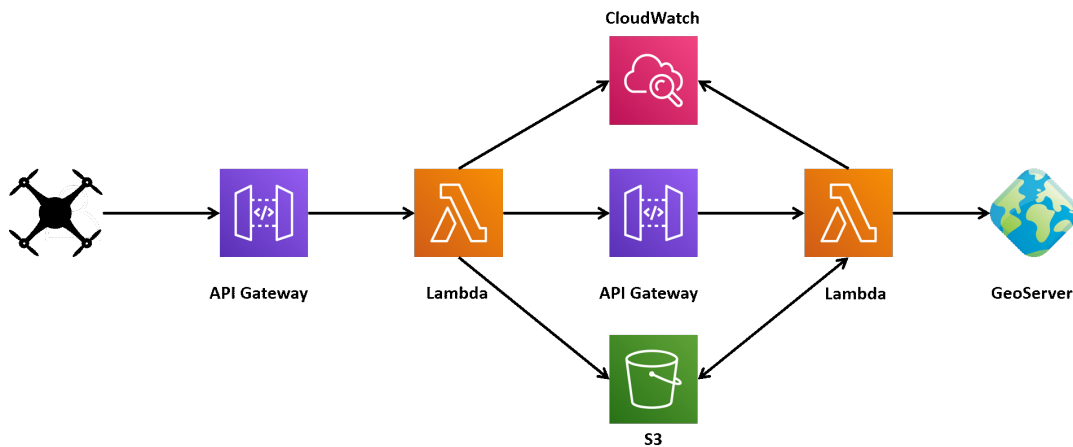


4.1.2 Progettazione architetturale

Come visto nel paragrafo 3.1, il progetto è stato sviluppato con il servizio AWS Lambda. Il linguaggio di programmazione utilizzato è Python, per via della sua flessibilità e portabilità. Il *runtime* utilizzato da AWS Lambda per Python contiene, di base, molti moduli aggiuntivi, tuttavia, per l'interpolazione dei dati è stata utilizzata la libreria GDAL (*Geospatial Data Abstraction Library*). La libreria in questione non è presente su AWS Lambda ed è stata aggiunta tramite l'utilizzo dei *Layer*, come spiegato nel sottoparagrafo 3.1.1.3.

Dall'analisi dei requisiti, è possibile individuare, concettualmente, due funzionalità principali da implementare: l'interpolazione dei dati e la comunicazione con GeoServer. Nella fase di implementazione, di conseguenza, si è diviso il codice in due funzioni Lambda.

Figura 4.2: Schematizzazione dell'applicazione



La prima funzione ha il compito di ricevere i dati tramite richiesta HTTP, interpolarli utilizzando la libreria GDAL e, infine, caricare il risultato sul servizio Amazon S3. La seconda funzione, invece, si occupa di prelevare l'immagine interpolata da S3, stabilire una connessione con GeoServer e fornirgli i parametri necessari per procedere alla pubblicazione del risultato.

4.2 Implementazione

Nei sottoparagrafi 4.2.1 e 4.2.2 le funzioni implementate vengono discusse in dettaglio.

4.2.1 LambdaInterpolator.py

```
1 from chalice import Chalice, Response
2 from osgeo import gdal
3 import json
4 import boto3
5 import time
6 import requests
7 import os
```

Snippet 4.1: Importazione dei moduli necessari

Inizialmente (*snippet* 4.1), vengono importati alcuni moduli aggiuntivi necessari:

- dal modulo **chalice** vengono importati **Chalice** e **Response**, il primo è il framework Python introdotto nel sottoparagrafo 3.1.1.6, il secondo, è un modulo che permette di personalizzare la risposta HTTP restituita;
- dal modulo **osgeo** viene importato **gdal**, il modulo contenente i “*Python bindings*” per la libreria C++ originaria [7], usato per l’interpolazione;
- **boto3** è un modulo che permette di interfacciare AWS Lambda con altri servizi AWS a livello di programmazione [4];
- il modulo **requests** permette di inviare richieste HTTP direttamente da Python in maniera semplice e concisa, usato per la comunicazione tra le funzioni Lambda e la *REST* API di GeoServer;
- per gestire il corpo (*payload*) delle richieste HTTP, sia in entrata che in uscita, viene usato il modulo **json**;
- i moduli **time** e **os** rivestono ruoli marginali.

```

1 app = Chalice(app_name="Lambda_Interpolator")
2
3 tmp_filepath = "/tmp/bugs.geojson"
4 timestr = time.strftime("%Y%m%d-%H%M%S")
5 tmp_object_name = "img/int_image_"

```

Snippet 4.2: Inizializzazione di variabili

Segue, nello *snippet* 4.2, la creazione di alcune variabili utili in seguito. In particolare, è possibile notare la creazione di una stringa *timestamp* usata per assegnare un nome semi-casuale al prodotto dell'interpolazione.

```

1 def put(file_to_upload, bucket_name="awsapp-storage"):
2
3     object_name = tmp_object_name + timestr + ".tiff"
4     s3 = boto3.client("s3")
5     try:
6         s3.upload_file(file_to_upload, bucket_name, object_name)
7         status = True
8         return status, object_name
9     except:
10        status = False
11        return status, False

```

Snippet 4.3: Funzione put()

La funzione `put()` (*snippet* 4.3) riceve il *path* di un file e il nome di un *bucket* come parametri. Utilizza, dunque, il modulo `boto3` per instaurare una connessione con il servizio S3 e caricare il file nel *bucket*, infine, restituisce la variabile booleana `status`. Quest'ultima fornisce informazioni sullo stato dell'operazione e, in fase di *testing*, permette di capire se l'upload su S3 è andato a buon fine.

```

1 def gdal_grid(path):
2
3     grid_opt = gdal.GridOptions(format="GTiff",
4                                outputType=gdal.GDT_Int16,
5                                algorithm="invdist:power=3.0:smoothing=0.0:
6                                radius1=0.0:radius2=0.0:angle=0.0:max_points=0:
7                                min_points=0:nodata=0.0",
8                                layers="bugs",
9                                zfield="value",
10                               )
11
12     # Use .tiff and not .tif
13     try:
14         output = gdal.Grid("/tmp/output.tiff", path, options=grid_opt)
15         status = True
16         return status
17     except:
18         status = False
19         return status

```

Snippet 4.4: Funzione gdal_grid()

Nello *snippet* 4.4 è mostrata la funzione chiave di `LambdaInterpolator.py`, `gdal_grid` infatti, si occupa dell'interpolazione dei dati tramite il modulo `gdal`.

Inizialmente, tramite la funzione `gdal.GridOptions` vengono impostati alcuni parametri necessari. Il parametro `algorithm` permette di impostare l'algoritmo da utilizzare per l'interpolazione, nel caso specifico, è stato utilizzato l'IDW (par.3.2.1.3) con un esponente $p = 3$. Un altro parametro importante è `radius`, porlo uguale a zero non implica impostare un raggio di ricerca nullo, ma indica all'algoritmo di effettuare la ricerca sull'intera matrice di punti.

Terminata la configurazione dei parametri, viene invocata `gdal.Grid`. Quest'ultima esegue effettivamente l'interpolazione, utilizzando i parametri impostati. L'output di tale funzione è un file con estensione `.tiff`, un formato immagine di tipo *raster*.

```

1 @app.route("/interpolate/{geojson}", methods=["POST"])
2 def interpolate(geojson):
3
4     if app.current_request.method != "POST":
5         return Response(body="Method not allowed, use POST!",
6                           status_code=405,
7                           headers={"Content-Type": "text/plain"})
8
9     headers = app.current_request.headers
10    content_type = headers["content-type"]
11
12    if content_type != "application/json":
13        return Response(body="Bad request, send .geojson file content as a json
14        body!",
15                          status_code=400,
16                          headers={"Content-Type": "text/plain"})
17
18    request = app.current_request
19    geojson = request.json_body
20    geojson = json.dumps(geojson)

```

Snippet 4.5: Funzione `interpolate()`

La funzione presentata nello *snippet* 4.5, oltre ad alcune operazioni basilari, funge da *pipeline*, ovvero ha il compito di invocare le altre funzioni descritte finora. Nella prima riga si nota il decoratore `@app.route()`, il suo utilizzo indica al framework `Chalice` che le richieste HTTP verso la risorsa `/interpolate` devono essere indirizzate all'omonima funzione.

Seguono dei controlli sul metodo della richiesta HTTP e sul contenuto del *payload*, respingendo le richieste che soddisfano i requisiti. Successivamente, il corpo della richiesta viene serializzato come stringa dalla funzione `json.dumps`.

```

1     f = open("/tmp/bugs.geojson", "w+")
2     f.write(geojson)
3     f.close()
4
5     int_status = gdal_grid(tmp_filepath)
6
7     if not int_status:
8         return "Interpolation failed!"

```

Snippet 4.6: Funzione `interpolate()`

I dati spaziali da interpolare sono codificati in formato `.geojson` [8] che, nonostante sia simile al formato `.json`, genera delle incompatibilità con la funzione `gdal.Grid` nel caso venga inviato, come corpo della richiesta, direttamente in tale formato. Per aggirare il problema, dopo aver serializzato il corpo della richiesta come stringa `json` (*snippet* 4.5), viene ricodificato in formato `.geojson` come mostrato nello *snippet* 4.6. Segue l'invocazione di `gdal_grid()`.

```

1     put_status, s3_path = put(file_to_upload="/tmp/output.tiff")
2
3     if not put_status:
4         return "Upload failed!"

```

Snippet 4.7: Funzione `interpolate()`

Terminata la funzione `gdal_grid()`, viene invocata la funzione `put()` che carica il risultato dell'interpolazione su S3 (*snippet* 4.7).

```

1     payload_dict = {"s3_path": s3_path, "timestr": timestr}
2     payload = json.dumps(payload_dict)
3
4     headers = {
5         "Content-Type": "application/json",
6         "cache-control": "no-cache"
7     }
8
9     url = os.environ.get("URL")
10
11    requests.request("POST", url, data=payload, headers=headers)

```

Snippet 4.8: Funzione `interpolate()`

Nello *snippet* 4.8, viene mostrata la fase finale della funzione `interpolate()`, nonché della funzione `Lambda`. Nello specifico, viene creato un `payload` contenente il `path` del file appena caricato su S3 e la stringa `timestamp`. Generati gli `headers` e recuperato l'URL della funzione `LambdaGeoServer`, viene effettuata una richiesta `POST` verso quest'ultima.

4.2.2 LambdaGeoServer.py

```

1 from chalice import Chalice
2 import requests
3 import json
4 import os

```

Snippet 4.9: Importazione dei moduli necessari

I moduli utilizzati dalla funzione `LambdaGeoServer` (*snippet* 4.9) sono stati già analizzati per `LambdaInterpolator` (*snippet* 4.1 par. 4.2.1).

```

1 def create_store(s3_path, timestr):
2
3     host = os.environ.get("HOST")
4     prefix = host + "/geoserver/rest/workspaces"
5     workspace_name = "univpm"
6     store_type = "coveragestores"
7
8     # Generate the url
9     url_tuple = (prefix, workspace_name, store_type)
10    url = "/".join(url_tuple)
11
12    # Generate the uri for a S3GeoTiff type store
13    region = "?awsRegion=EU_CENTRAL_1"
14    s3_uri = "s3://awsapp-storage" + s3_path + region
15
16    store_name = "store" + timestr

```

Snippet 4.10: Funzione `create_store()`

Nello *snippet* 4.10 è mostrata la funzione `create_store()`.

Uno *store* GeoServer è un oggetto che si connette ad una sorgente dati. Una sorgente dati è, tipicamente, una tabella di un database, un singolo file *raster* o una *directory* contenente più file. Il vantaggio nell'utilizzo di uno *store* risiede nell'impostare i parametri di connessione alla sorgente dati una sola volta, con la possibilità di riutilizzare lo stesso *store* per più *dataset*.

Nel caso in esame, per usare come sorgente dati un file ospitato su S3, è stato necessario installare un *plugin* aggiuntivo [14]. Tale *plugin* presenta un inconveniente, ovvero non permette di usare il medesimo *store* per più file, dato che il suo parametro di connessione accetta soltanto uno specifico *Uniform Resource Identifier* (URI). L'URI è un parametro necessario al servizio S3 per identificare univocamente una risorsa.

La funzione, dunque, prende in ingresso il *path* S3 dove è stato caricato il file in precedenza e la stringa *timestamp* generata nella funzione `LambdaInterpolator`. La stringa contenente il *timestamp* è necessaria per recuperare l'immagine corretta dalla *directory* S3. Il nome delle immagini interpolate, infatti, è composto dallo stesso prefisso (`int_image_`) più il suffisso *timestamp*, generando nomi composti dal prefisso e dalla data di creazione del file. Genera, quindi, l'URI del file ed assegna un nome allo *store*.

```

1     payload_dict = {
2         "coverageStore": {
3             "name": store_name,
4             "enabled": "true",
5             "type": "S3GeoTiff",
6             "workspace": workspace_name,
7             "url": s3_uri } }
8
9     payload = json.dumps(payload_dict)

```

Snippet 4.11: Funzione `create_store()`

Con riferimento allo *snippet* 4.11, la funzione `create_store()` prosegue con la creazione del *payload* da inviare a GeoServer. Serializzato il corpo della richiesta in formato `json`, si procede a generare gli *headers* ed eseguire una richiesta `POST` verso la *REST* API di GeoServer, a questo punto la creazione dello *store* è completata (*snippet* 4.12).

```

1  headers = {
2      "Content-Type": "application/json",
3      "Authorization": "Basic YWRtaW46Z2Vvc2VydmVy",
4      "Accept": "*/*",
5      "Cache-Control": "no-cache",
6      "Host": host,
7      "Accept-Encoding": "gzip, deflate",
8      "Connection": "keep-alive",
9      "cache-control": "no-cache"
10 }
11
12 requests.request("POST", url, data=payload, headers=headers)
13
14 return store_name

```

Snippet 4.12: Funzione `create_store()`

La fase successiva è la creazione di un nuovo *layer*. In GeoServer, il termine *layer*, è usato per indicare un *dataset* di tipo raster o vettoriale. Nel progetto in questione, un *layer* rappresenta l'immagine interpolata.

Per concludere, al *layer* creato viene applicato uno "stile" (*style*). In GeoServer, uno stile contiene informazioni sull'aspetto da assegnare ad un *layer*.

Le funzioni utilizzate per la creazione di un *layer* e per l'assegnazione dello stile vengono, per completezza, presentate negli *snippet* 4.13 e 4.14. Tuttavia, non sono analizzate in dettaglio, essendo il loro funzionamento del tutto analogo alla funzione `create_store()`.

```

1  def publish_layer(s3_path, store):
2
3      .
4      .
5      .
6
7      requests.request("POST", url, data=payload, headers=headers)
8
9      return layer_name

```

Snippet 4.13: Funzione `publish_layer()`

```

1  def set_style(layer_name):
2
3      .
4      .
5      .
6
7      requests.request("POST", url, params=query_params,
8                      data=payload, headers=headers)
9
10     return

```

Snippet 4.14: Funzione `set_style()`

4.3 Testing

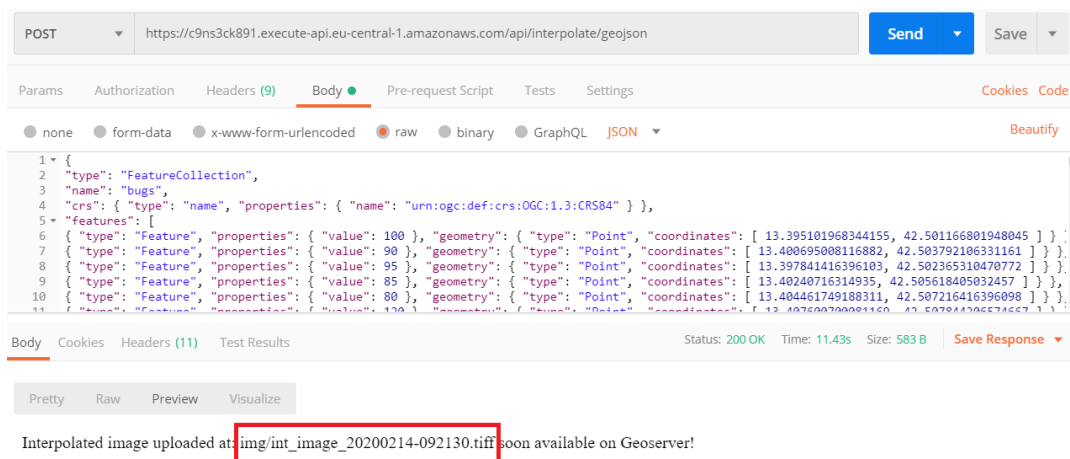
Nel presente paragrafo viene mostrato il funzionamento effettivo dell'applicazione *serverless* sviluppata. Nel sottoparagrafo 4.3.1 viene presentato il processo di interpolazione dei dati, mentre nel sottoparagrafo 4.3.2 viene mostrato il risultato finale pubblicato su GeoServer.

Si ricorda che il processo di interpolazione e pubblicazione è completamente automatico. Tuttavia, al fine di mostrare singolarmente il risultato di ogni operazione, le varie richieste HTTP sono effettuate manualmente con l'ausilio di Postman [15].

4.3.1 Interpolazione dei dati

Inizialmente, viene effettuata la richiesta POST per inviare i dati da interpolare sotto forma di `.json`. In figura 4.3 è mostrato il risultato della richiesta, inoltre, è evidenziato in rosso il nome assegnato all'immagine interpolata. La figura 4.4 mostra, invece, la medesima immagine interpolata, caricata su S3.

Figura 4.3: Richiesta di interpolazione dei dati



The screenshot shows a Postman interface for a POST request to `https://c9ns3ck891.execute-api.eu-central-1.amazonaws.com/api/interpolate/geojson`. The response is a JSON object:

```

1 {
2   "type": "FeatureCollection",
3   "name": "bugs",
4   "crs": { "type": "name", "properties": { "name": "urn:ogc:def:crs:OGC:1.3:CRS84" } },
5   "features": [
6     { "type": "Feature", "properties": { "value": 100 }, "geometry": { "type": "Point", "coordinates": [ 13.395101968344155, 42.501166801948045 ] } },
7     { "type": "Feature", "properties": { "value": 90 }, "geometry": { "type": "Point", "coordinates": [ 13.400695088116882, 42.503792106331161 ] } },
8     { "type": "Feature", "properties": { "value": 95 }, "geometry": { "type": "Point", "coordinates": [ 13.397841416396103, 42.502365310470772 ] } },
9     { "type": "Feature", "properties": { "value": 85 }, "geometry": { "type": "Point", "coordinates": [ 13.40240716314935, 42.505618405032457 ] } },
10    { "type": "Feature", "properties": { "value": 80 }, "geometry": { "type": "Point", "coordinates": [ 13.404461749188311, 42.507216416396098 ] } },
11    { "type": "Feature", "properties": { "value": 100 }, "geometry": { "type": "Point", "coordinates": [ 13.407600700081160, 42.507841406574662 ] } }
  ]
}

```

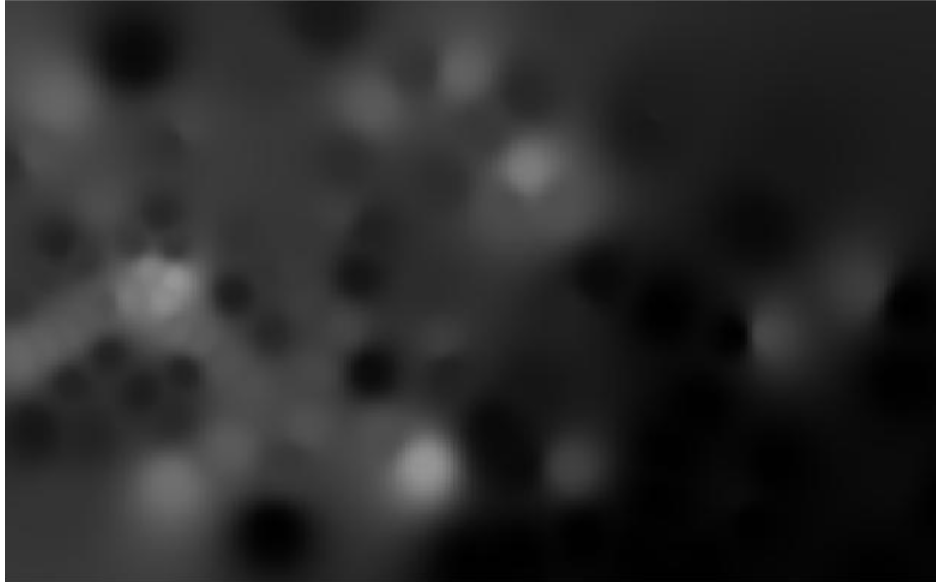
The response status is 200 OK, Time: 11.43s, Size: 583 B. Below the response, the text "Interpolated image uploaded at `img/int_image_20200214-092130.tiff` soon available on Geoserver!" is shown, with the filename highlighted in red.

Figura 4.4: Risultato dell'interpolazione caricato su S3

Nome	Ultima modifica	Dimensioni
<code>int_image_20200214-092130.tiff</code>	feb 14, 2020 10:21:40 AM GMT+0100	128.5 KB
<code>int_image_20191216-130755.tiff</code>	dic 16, 2019 2:08:05 PM GMT+0100	128.5 KB
<code>int_image_20191216-123428.tiff</code>	dic 16, 2019 1:34:38 PM GMT+0100	128.5 KB
<code>int_image_20191212-223058.tiff</code>	dic 12, 2019 11:31:08 PM GMT+0100	128.5 KB
<code>test_image.tiff</code>	dic 12, 2019 11:23:10 PM GMT+0100	128.5 KB

Il risultato dell'interpolazione è mostrato in figura 4.5.

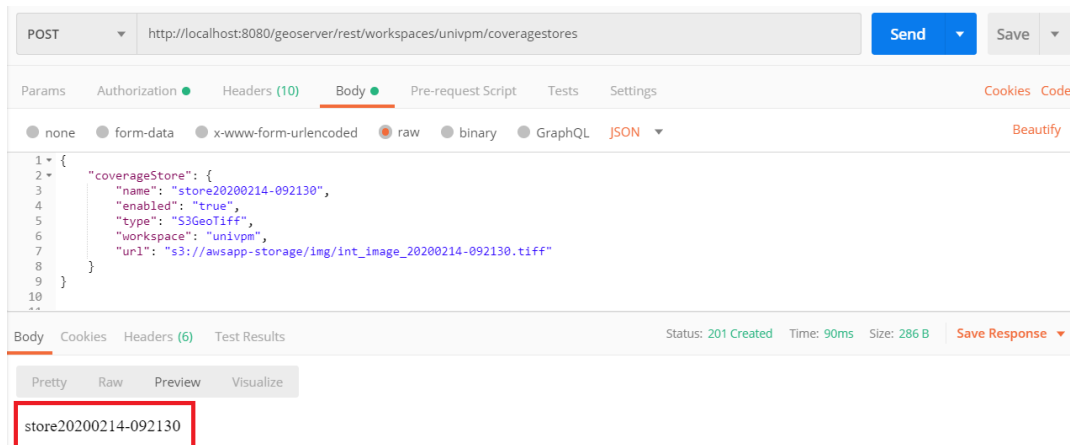
Figura 4.5: Risultato dell'interpolazione



4.3.2 Pubblicazione su GeoServer

Come spiegato nel sottoparagrafo 4.2.2, per procedere alla pubblicazione su GeoServer, il primo passo è creare uno *store*, come mostrato in figura 4.6.

Figura 4.6: Richiesta di creazione di uno store



La figura 4.7 mostra, invece, il risultato della richiesta nell'interfaccia grafica di GeoServer.

Figura 4.7: Risultato della creazione dello store

<input type="checkbox"/>		sf	sfdem	GeoTIFF	✓
<input type="checkbox"/>		topp	states_shapefile	Shapefile	✓
<input type="checkbox"/>		univpm	store20191212-223058	S3GeoTiff	✓
<input type="checkbox"/>		univpm	store20200214-092130	S3GeoTiff	✓
<input type="checkbox"/>		univpm	store_name	GeoTIFF	✓

In figura 4.8 seguono, dunque, le richieste per pubblicare il *layer* e applicare lo stile. La figura 4.9 mostra l'effettiva pubblicazione in GeoServer.

Figura 4.8: Richiesta di pubblicazione del Layer e impostazione dello stile

The first request is a POST to `http://localhost:8080/geoserver/rest/workspaces/univpm/coveragestores/store20200214-092130/coverages`. The body is a JSON object:

```

1 {
2   "coverage": {
3     "name": "int_image_20200214-092130",
4     "nativeName": "int_image_20200214-092130"
5   }
6 }

```

The second request is a POST to `http://localhost:8080/geoserver/rest/layers/int_image_20200214-092130/styles?default=true`. The body is a JSON object:

```

1 {
2   "style": {
3     "name": "univpm_style"
4   }
5 }

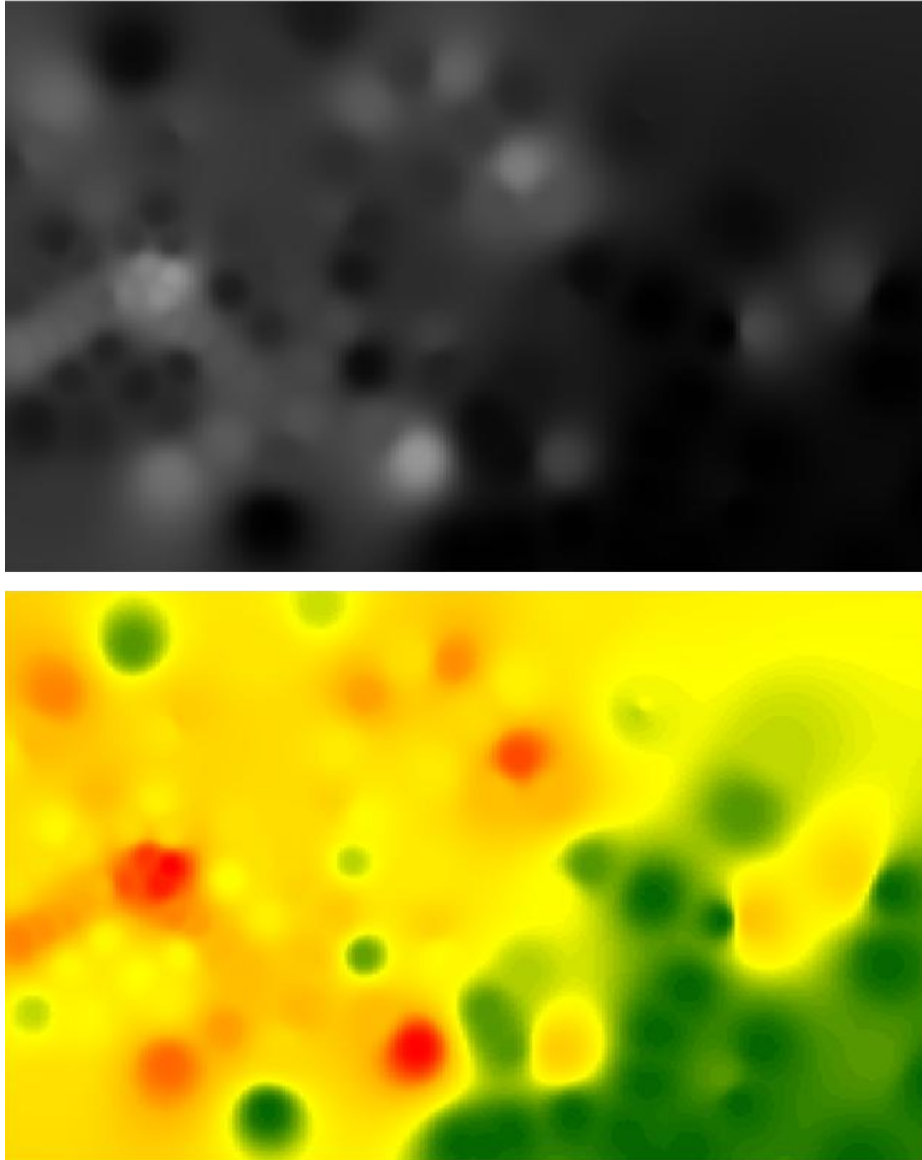
```

Figura 4.9: Risultato della pubblicazione del Layer

Type	Title	Name	Common Formats	All Formats
	int_image_20191212-223058	univpm:int_image_20191212-223058	OpenLayers KML	Select one
	int_image_20200214-092130	univpm:int_image_20200214-092130	OpenLayers KML	Select one
	test	univpm:test	OpenLayers KML	Select one
	int_image_20191212-223058	univpm:int_image_20191212-223058	OpenLayers KML	Select one
	test	univpm:test	OpenLayers KML	Select one
	int_image_20191212-223058	univpm:int_image_20191212-223058	OpenLayers KML	Select one
	int_image_20191212-223058	univpm:int_image_20191212-223058	OpenLayers KML	Select one

In conclusione, in figura 4.10 viene mostrato il confronto tra l'interpolazione "grezza" e la successiva applicazione dello stile.

Figura 4.10: In alto: l'immagine risultante dall'interpolazione.
In basso: l'immagine con lo stile applicato.



Capitolo 5

Conclusioni e Sviluppi Futuri

5.1 Conclusioni

Il presente elaborato ha illustrato l'intero processo di sviluppo del progetto di tesi, partendo dallo studio preliminare delle tecniche e degli strumenti necessari, fino all'implementazione pratica.

L'obiettivo era automatizzare un processo di interpolazione di dati geografici utilizzando le tecnologie *Serverless*. Le tecniche e gli strumenti impiegati hanno permesso di rispettare tutti i requisiti progettuali. In particolare, la scelta del linguaggio di programmazione Python, ha garantito un'ottima produttività e flessibilità. Python, infatti, garantisce un buon compromesso tra semplicità e potenzialità offerte, soprattutto grazie al supporto di numerosi moduli aggiuntivi che ne estendono le funzionalità.

La piattaforma cloud Amazon Web Services ha, inoltre, garantito le giuste prestazioni durante l'elaborazione e, grazie alla sua interfaccia semplice e intuitiva, ha permesso di concentrarsi fin da subito sullo sviluppo dell'applicazione.

Il servizio AWS Lambda ha garantito una semplice gestione del codice. Inoltre, l'utilizzo dei Lambda *Layer* ha permesso di sfruttare la libreria GDAL che, altrimenti, ha dimensioni superiori al massimo consentito da AWS Lambda. In aggiunta, il servizio API Gateway, ha permesso di gestire con estrema facilità la comunicazione tra i vari servizi utilizzati facilitando lo sviluppo di API REST.

In conclusione, nonostante ci siano margini di miglioramento, lo sviluppo dell'applicazione *serverless* ha soddisfatto tutti i requisiti funzionali richiesti.

5.2 Sviluppi futuri

I metodi e le tecniche implementate nello sviluppo dell'applicazione, pur avendo permesso di raggiungere gli obiettivi imposti, possono risultare non adatte a casi di studio più complessi.

In particolare, una volta pubblicato il risultato dell'interpolazione, il ciclo di vita dell'applicazione termina, rendendo l'elaborazione fine a se stessa. Una possibilità è quella di implementare nuove funzionalità nel campo della statistica. Ciò permetterebbe di generare tabelle, grafici, diagrammi e previsioni sull'andamento della popolazione di insetti nell'area di studio.

Un'altra funzionalità utile è l'aggiunta di un'interfaccia grafica per gestire in modo semplice e intuitivo l'upload dei dati e l'impostazione dei parametri di interpolazione.

Bibliografia

- [1] Boufassa A. and Armstrong M. *Comparison between different kriging estimators*. Mathematical Geology, 1989.
- [2] <https://aws.amazon.com/api-gateway/features>
ultima visita 20/01/2020.
- [3] <https://aws.amazon.com/what-is-aws>
ultima visita 22/01/2020.
- [4] <https://boto3.amazonaws.com/v1/documentation/api/latest/index.html>
ultima visita 08/02/2020.
- [5] <https://github.com/aws/chalice>
ultima visita 10/02/2020.
- [6] <https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2-beta/>
ultima visita 23/01/2020.
- [7] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation, 2019.
- [8] <https://en.wikipedia.org/wiki/geojson>
ultima visita 13/02/2020.
- [9] <http://geoserver.org/about/>
ultima visita 12/02/2020.
- [10] Danie Krige. *A statistical approach to some basic mine valuation problems on the Witwatersrand*. Metal. and Mining Soc. of South Africa, 1951.
- [11] <https://aws.amazon.com/lambda/features>
ultima visita 20/01/2020.
- [12] Jin Li and Andrew D. Heap. A review of spatial interpolation methods for environmental scientists. *Geoscience Australia*, 2008.

- [13] https://en.wikipedia.org/wiki/nearest-neighbor_interpolation
ultima visita 11/02/2020.
- [14] <https://docs.geoserver.org/latest/en/user/community/s3-geotiff/index.html>
ultima visita 03/02/2020.
- [15] <https://www.postman.com/>
ultima visita 13/02/2020.
- [16] <https://www.qgis.org/it/site/about/index.html>
ultima visita 12/02/2020.
- [17] Antonio Regalado. Who coined “cloud computing”. *Technology Review*, 2011.
- [18] <https://aws.amazon.com/s3/features>
ultima visita 15/01/2020.
- [19] https://en.wikipedia.org/wiki/triangulated_regular_network
ultima visita 10/02/2020.
- [20] Hans Wackernagel. *Multivariate Geostatistics: An Introduction with Applications*. Springer, 2003.
- [21] Richard Webster and Margaret Oliver. *Geostatistics for Environmental Scientists*. John Wiley & Sons, Ltd, 2001.

Elenco delle figure

2.1	Livelli di astrazione del <i>Cloud Computing</i>	8
2.2	Schema approccio serverless	9
2.3	Logo Amazon AWS	10
2.4	Esempio di trigger in Lambda	11
2.5	Interazioni di API Gateway	11
2.6	Integrazione di S3 in altri servizi AWS	12
3.1	Console di gestione AWS	15
3.2	Pannello di riepilogo IAM	16
3.3	Chiave d'accesso IAM	16
3.4	Riepilogo bucket S3	17
3.5	Cartella S3	17
3.6	Proprietà di un file su S3	18
3.7	Lista funzioni Lambda	18
3.8	Designer Lambda	19
3.9	Console per la modifica del codice	20
3.10	Configurazione variabili d'ambiente su Lambda	20
3.11	Pagina di configurazione di API Gateway	21
3.12	Lista gruppi di log su CloudWatch	22
3.13	Dettaglio di un log	22
3.14	Esempio di deploy con Chalice	23
3.15	Testing in locale con Chalice	23
3.16	Generazione di dati spaziali con QGIS	24
3.17	Esempio di Layer pubblicato su GeoServer	24
3.18	Esempio di poligoni di Voronoi	25
3.19	Esempio di rete irregolare di triangoli	26
3.20	Triangolazione di Delaunay	26
3.21	Esempio di interpolazione con IDW al variare di p	27
3.22	Esempio di semivariogramma	28
4.1	Schematizzazione dei requisiti	31
4.2	Schematizzazione dell'applicazione	32
4.3	Richiesta di interpolazione dei dati	39

4.4	Risultato dell'interpolazione caricato su S3	39
4.5	Risultato dell'interpolazione	40
4.6	Richiesta di creazione di uno store	40
4.7	Risultato della creazione dello store	41
4.8	Richiesta di pubblicazione del Layer e impostazione dello stile . .	41
4.9	Risultato della pubblicazione del Layer	41
4.10	Confronto tra l'interpolazione originale e l'applicazione dello stile	42