Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso a Libera Scelta - 6 CFU

Cloud Computing

Professore:
Prof. Antonio Brogi

Autore:
Filippo Ghirardini

Anno Accademico 2023/2024

Contents

Cloud Computing

Realizzato da: Ghirardini Filippo

A.A. 2023-2024

---

1 Introduzione

1.1 Service-based economy

L'economia basata sui servizi (*Everything as a service*) si fonda sulla tendenza degli ultimi 30 anni di passare dai *beni* ai *servizi*.

Esempio 1.1.1 (Bicicletta)**.** Una signora compra una bicicletta dal venditore. Questa diventa di sua proprietà e si deve occupare della manutenzione. La bicicletta diventa un **servizio** (*CicloPi*) e la signora paga per usare la bici che però non è più sua (e non deve più preoccuparsi di manutenzione e furto).

Esempi più vicini all'informatica sono il passaggio dai supporti fisici per la musica allo streaming o i dispositivi di memorizzazione passati ai Cloud Drive.

1.2 Service contracts

Quando usiamo un servizio non vogliamo sapere come viene implementato. L'unica cosa che ci interessa è cosa è specificato sul **contratto di utilizzo**. Nella maggior parte dei casi l'utente non lo legge.

1.2.1 Quality of Service

Ci sono più fornitori che ci danno lo stesso servizio ma con qualità del servizio diverse. Dobbiamo chiederci se il prezzo più basso vale la pena del sacrificio della qualità.

1.2.2 Service Level Agreement

Sono i contratti di servizio che includono anche il livello di **affidabilità di servizio**. In questa situazione abbiamo tre figure:

- *Programmatore*
- *Business expert*: colui che sa il livello di affidabilità in base al mercato
- *Legale*: colui che sa come scriverlo

Esempio 1.2.1 (Google SLA)**.** Google Compute Engine fornisce un **Service Level Objective** (SLO) del 99.95%. In caso di non raggiungimento del SLO si viene rimborsati con del credito in percentuale a quanto si è distanti dal target.

Montlhy Uptime Percentage	**Rimborso**
$95.00\% - < 99.95\%$	10%
$90.00\% - < 95.00\%$	25%
$< 90.00\%$	100%

Se ad esempio l'1 e il 2 Aprile dalle 8am alle 5pm (orario di lavoro) non era disponibile il servizio, a quanto ammonta il rimborso in credito?
Dobbiamo prima capire cos'è il **Montly Uptime Percentage** secondo l'SLA:

Definizione 1.2.1 (Monthly Uptime Percentage)**.** *Total number of minutes in a month, minus the number of minutes of* ***Downtime*** *suffered from all* ***Downtime Period*** *in a month, divided by the total number of minutes in a month.*

Definizione 1.2.2 (Downtime Period)**.** *A period of one of more consecutive minutes of* ***Downtime****.*

Definizione 1.2.3 (Downtime)**.** *For virtual machines instances: loss of external connectivity or persistent disk accessfor the Single Instance or, with respect to Instances in Multiple Zones, all applicable running instances.*

Inoltre il cliente deve richiedere entro 60 giorni il rimborso fornendo anche dei **log file** che mostrino il **Downtime Period** assieme alla data e all'orario.

a

Definizione 1.2.4 (Legge del pesce rosso)**.** *Quando un fornitore di servizi non garantisce in alcun modo il prodotto.*

Esempio 1.2.2 (Microsoft Service Agreement)**.** Microsoft non è responsabile per alcuna perdita di dati o interruzione di servizi e non li garantisce in alcun modo (vedi art.6 e art.11).

Esempio 1.2.3. Supponiamo di avere un servizio che si appoggia a due servizi cloud, ognuno disponibile in maniera indipendente il 90% del tempo. Il servizio è probabile che non sarà disponibile per 4 ore e mezza al giorno:

$$(1 - (0.90 * 0.90)) * 24 = 4.56$$

1.3 Cloud

Un primo fattore da tenere in considerazione è la **service demand**, che cambia nel tempo.
Prima del cloud c'era il problema di dimensionare il servizio in base a delle stime, che possono essere di due tipi:

- **Overprovisioning**: utilizzare un infrastruttura che possa sopportare anche i carichi massimi previsti. In questo caso abbiamo un problema di **spreco di risorse**.
- **Underprovisioning**: quando l'infrastruttura non è sufficiente per tutti i momenti di carico. Questo causerebbe la perdita di clienti che dopo qualche tentativo fallito di utilizzo abbandonerebbero il servizio.

Il **cloud** ci fornisce risorse apparentemente infinite disponibili su richiesta.

Definizione 1.3.1 (Cloud)**.** *Secondo il NIST il Cloud Computing è un **modello** per consentire l'accesso ubiquo, conveniente e a richiesta di risorse computazionali.*
Le idee chiave sono le seguenti:

- *Messa in comune di strutture autogestite, utilizzate come servizi*
- *Risorse virtuali **scalabili** disponibili attraverso internet a diversi clienti*
- ***Separazione** dei servizi dalla tecnologia alla base*

*Dal punto di vista economico sfrutta il principio **pay-per-use**, che permette all'utente di convertire i costi da Capital Expenses a Operation Expenses. Questo garantisce elasticità e trasferimento dei rischi al fornitore dei servizi.*

1.3.1 Service models

Analizziamo ora i modelli di servizio, prendendo come esempio la pizza:

- **IaaS**: fornisce server virtualizzati, archiviazione e rete e li gestisce. Il cliente è responsabile per tutti gli altri aspetti, quali OS e applicazioni. Nel nostro caso sarebbe comprare una pizza da cuocere.
- **PaaS**: fornisce l'intera piattaforma come servizio (VM, OS, servizi, SDKs) e gestisce l'infrastruttura, l'OS e l'abilitazione del software. Il cliente è responsabile dell'installazione e la gestione delle applicazioni. Nel nostro caso sarebbe una pizza d'asporto.
- **SaaS**: fornisce software a richiesta, disponibile tramite API. Viene gestita l'infrastruttura, l'OS e l'applicazione, quindi il cliente non è responsabile di nulla. Nel nostro caso sarebbe la pizzeria.

Vediamo il confronto in quanto a successo *revenue* e *market share*:

1.3.2 Deployments models

La scelta del deployment model può essere **pubblico**, **privato** o **ibrido**. Il modello privato garantisce un maggiore controllo sui dati mentre quello pubblico una maggiore scalabilità. La versione ibrida divide i dati tra pubblico e privato in base alle esigenze di sicurezza e scalabilità.

Tipo	1H22 Rev	1H23 Rev	1H22 Msh	1H23 Msh
IaaS	55.1 BUSD	64.4 BUSD	20.8%	20.4%
PaaS	44.5 BUSD	56.8 BUSD	16.8%	18.0%
SaaS SW	121.9 BUSD	141.2 BUSD	46.0%	44.7%
SaaS HW	43.3 BUSD	53.2 BUSD	16.4%	16.9%

1.3.3 Domande

Ci si deve fare alcune domande sulla confidenzialità dei nostri dati:

- Dove saranno fisicamente conservati i nostri dati?
- La privacy e l'integrità dei nostri dati sono garantite? Come?
- Come possiamo sapere se c'è stato un problema?

sulla disponibilità dei servizi:

- Cosa succede se il fornitore del servizio non lo fornisce?
- SPoF (Single Point of Failure): quando la nostra architettura sfrutta un solo servizio che potrebbe fallire

Le risposte le troviamo tutte nel SLA.

1.3.4 Vendor Lock-In

Definizione 1.3.2 (Vendor Lock-In)**.** *Il Vendor Lock-In è quando si rende un cliente dipendente da un venditore per prodotti o servizi, rendendogli impossibile il cambio ad un altro gestore senza affrontare costi esorbitanti.*

Cosa succede se si vuole cambiare provider? Il software continuerà a funzionare? Si potrà trasferire facilmente i propri dati? Serve sempre un **exit plan**.

2 Container

I container sono un metodo per creare la virtualizzazione e l'isolamento delle risorse in maniera più semplice.
La struttura è simile a quella della virtualizzazione, dove al posto dell'hypervisor abbiamo un **container manager**. La differenza principale è, però, che tutti i container operano sopra ad un unico OS, permettendo di risparmiare molte risorse sia in termini di performance che di spazio.

2.1 Storia

- UNIX **chroot** permetteva isolamento dal filesystem
- FreeBSD **jail** estese l'isolamento di chroot ai processi
- Google inizia a sviluppare i **CGroups** per il kernel di Linux
- I **Linux Container** forniscono una soluzione completa
- **Docker** aggiunge il concetto di immagini portabili e grafica semplice da utilizzare

2.2 Docker

Docker è una piattaforma che permette di sviluppare ed eseguire applicazioni in un ambiente isolato sfruttando i container.
I componenti software sono gestiti come **immagini** in sola lettura su cui vengono creati ed eseguiti i **container**. In aggiunta possono essere montati dei **volumi** per garantire la persistenza dei dati.

2.2.1 Images

Sono in sola lettura e vengono salvate in un **docker registry**, pubblico o privato, strutturati in repositories che contengono ognuna ogni versione di un determinato software.

2.2.2 Swarm mode

Docker prevede la **swarm mode** per gestire un gruppo di container (swarm). I **nodi** possono essere:

- *Managers*, che delegano le task ai workers
- *Workers*, che eseguono le task a loro assegnate

L'utente può definire lo stato desiderato dei vari servizi dell'applicazione. Ogni nodo avrà un DNS name univoco. Swarm si occuperà di fare *load balancing* e di mantenere la coerenza degli stati secondo quello desiderato.

3 FaaS

L'approccio FaaS è un mercato in crescita e ad oggi esistono molteplici opzioni tra le quali scegliere. In particolare le suddividiamo in due categorie: *commerciali* (e.g. AWS Lambda) e *open source* (e.g. Apache Openwisk).
Per fare una scelta consapevole, dobbiamo analizzare le proposte da due punti di vista: quello commerciale e quello tecnico.

3.1 Business view

3.1.1 Licensing

Tutte le piattaforme **open source** usano licenze abbastanza permissive, come ad esempio Apache 2.0. Quelle **commerciali** invece usano software proprietario, fatta eccezione per MS Azure Functions che ha alcune componenti libere.

	License	Type
Apache Openwhisk	Apache 2.0	Permissive
AWS Lambda	AWS service terms	Proprietary
Fission	Apache 2.0	Permissive
Fn	Apache 2.0	Permissive
Google Cloud Functions	Google Cloud platform terms	Proprietary
Knative	Apache 2.0	Permissive
Kubeless	Apache 2.0	Permissive
MS Azure Functions	SLA for Azure Functions	Proprietary
Nuclio	Apache 2.0	Permissive
OpenFaaS	MIT	Permissive

3.1.2 Installazione

Tra le opzioni commerciali solamente Azure functions ha dei servizi installabili *on-premises*. Quelle open source supportano invece molteplici host, con Kubernetes tra i più popolari.

	Type	Target hosts
Apache Openwhisk	Installable	Docker, Kubernetes, Linux, MacOS, Mesos, Windows
AWS Lambda	as-a-service	n/a
Fission	Installable	Kubernetes
Fn	Installable	Docker, Linux, MacOS, Unix, Windows
Google Cloud Functions	as-a-service	n/a
Knative	Installable	Kubernetes
Kubeless	Installable	Kubernetes, Linux, MacOS, Windows
MS Azure Functions	as-a-service, installable	Linux, Kubernetes, MacOS, Windows
Nuclio	as-a-service, installable	Kubernetes
OpenFaaS	Installable	Docker[3], faasd, Kubernetes, OpenShift

3.1.3 Source code

Tutte le piattaforme open source sono hostate su GitHub ed implementate in Go.

	Availability	Open source repository	Programming language
Apache Openwhisk	Open source	GitHub	Scala
AWS Lambda	Closed source	n/a	n/a
Fission	Open source	GitHub	Go
Fn	Open source	GitHub	Go
Google Cloud Functions	Closed source	n/a	n/a
Knative	Open source	GitHub	Go
Kubeless	Open source	GitHub	Go
MS Azure Functions	Open source[a]	GitHub	C#
Nuclio	Open source	GitHub	Go
OpenFaaS	Open source	GitHub	Go

3.1.4 Interfaccia

Tutte le piattaforme forniscono CLI, mentre API e GUI non sono sempre presenti in quelle open source. Inoltre in queste ultime il metodo di amministrazione cambia molto tra l'una e l'altra.

		Apache Openwhisk	AWS Lambda	Fission	Fn	Google Cloud Functions	Knative	Kubeless	MS Azure Functions	Nuclio	OpenFaaS
Type	cli	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	api	✓	✓	✓	✓	✓	✓	×	✓	×	✓
	gui	×	✓	×	✓	✓	×	×	✓	✓	✓
App. Man.	create	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	retrieve	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	update	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	delete	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Plat. Adm.	deployment	✓	×	✓	✓	×	✓	✓	×	✓	✓
	configuration	✓	×	✓	✓	×	✓	✓	×	✓	×
	enactment	✓	×	✓	✓	×	✓	✓	×	✓	✓
	termination	×[a]	×	×[a]	✓	×	×[a]	×[a]	×	×[a]	×[a]
	undeployment	×[a]	×	×[a]	×[a]	×	×[a]	×[a]	×	×[a]	×[a]

[a]Termination/undeployment can be achieved by stopping/uninstalling the platform instance with host commands.

3.1.5 Community

Le piattaforme open source sono molto ben votate su GitHub ma poco cercate e diffuse su StackOverflow, a differenza di quelle commerciali.

		Apache Openwhisk	AWS Lambda	Fission	Fn	Google Cloud Functions	Knative	Kubeless	MS Azure Functions	Nuclio	OpenFaaS
GitHub	Stars	4.8K	n/a	5.2K	4.6K	n/a	3K[a]	5.8K	n/a	3.3K	17.9K
	Forks	932	n/a	487	349	n/a	637[a]	591	n/a	332	1.5K
	Issues	274	n/a	215	125	n/a	223[a]	164	n/a	51	62
	Commits	2.8K	n/a	1.2K	3.4K	n/a	4.7K[a]	1K	n/a	1.4K	1.9K
	Contributors	180	n/a	104	86	n/a	185[a]	89	n/a	55	147
SO	Questions	198	16.8K	7	25	10.1K	71	8	7.2K	3	29

[a]Values for the function hosting component of Knative, i.e., Knative Serving.

3.1.6 Documentazione

Tutte le piattaforme forniscono la documentazione per l'utilizzo e il deploy del servizio ma non tutte le loro architetture sono documentate.

		Apache Openwhisk	AWS Lambda	Fission	Fn	Google Cloud Functions	Knative	Kubeless	MS Azure Functions	Nuclio	OpenFaaS
App.	Development	✓	✓	✓	✓	✓	×	✓	✓	×	×
	Deployment	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Platform	Usage	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Development	✓	×	✓	×[a]	×	✓	✓	×	×[a]	✓
	Deployment	✓	×	✓	✓	×	✓	✓	✓	✓	✓
	Architecture	✓	×	✓	✓	×	×	✓	×	✓	✓

[a]Only providing guidelines/code of conduct for contributing to the project.

3.2 Technical view

3.2.1 Sviluppo

I linguaggi più comuni nello sviluppo in ambito FaaS sono Java, NodeJS e Python, con il supporto di Docker.

Classification of considered FaaS platforms, based on the *Development* category in the *technical* view of our classification framework. D and E are used to denote that quotas are set for *deployment package size* and *execution time*, respectively. The abbreviation "n/s" stays for "not specified", meaning that no related information is in the documentation.

	IDEs and Text Editors	Client Libraries	Quotas
Apache Openwhisk	Visual Studio Code[a], Xcode[a]	Go, Node.js	E[b]
AWS Lambda	AWS Cloud9, Eclipse, Toolkit for JetBrains, Visual Studio, Visual Studio Code	Go, Java, MS .NET, Node.js, Python, Ruby, C++	D, E
Fission	n/s	n/s	n/s
Fn	n/s	Go	n/s
Google Cloud Functions	n/s	Dart, Go, Java, MS .NET, Node.js, Python, Ruby	D, E
Knative	n/s	n/s	n/s
Kubeless	Visual Studio Code	n/s	n/s
MS Azure Functions	Visual Studio, Visual Studio Code	Java, MS .NET, Python	D, E[b]
Nuclio	Jupyter Notebooks	Go, Java, MS .NET, Python	n/s
OpenFaaS	n/s	n/s	n/s

[a]Deprecated/No longer maintained.
[b]Bounded by default, but can be configured to unset quota.

3.2.2 Versioning

La maggior parte delle piattaforme implementano un sistema di versioning in maniera implicita, al contrario di quelle commerciali:

- **Dedicated mechanisms**
- **Implicit mechanisms**

Classification of considered FaaS platforms, based on the *Version Management* category in the *technical* view of our classification framework. D and I are used to denote the possible values *dedicated mechanisms* and *implicit versioning*, respectively. The abbreviation "n/s" stays for "not specified", meaning that a platform does not explicitly mention the versioning of serverless applications.

	Application versions	Function versions
Apache Openwhisk	I	I
AWS Lambda	D	D
Fission	I	I
Fn	I	D
Google Cloud Functions	I	I
Knative	n/s	D
Kubeless	n/s	I
MS Azure Functions	I	I
Nuclio	n/s	D
OpenFaaS	n/s	I

3.2.3 Event sources

Tutte le piattaforme supportano la chiamata di funzioni tramite richieste **sincrone** HTTP, mentre solo alcune supportano quelle **asincrone**. Più della metà non supportano il salvataggio dei **dati delle richieste**. Gli **scheduler**, gli **stream processing** e il **messaging** sono ampiamente supportati. Più della metà delle piattaforme documentano la possibilità di creare **eventi personalizzati**.

3.2.4 Function orchestration

Più della metà delle piattaforme supportano il function orchestration tramite DSL proprietari o funzioni apposite.

Classification of considered FaaS platforms, based on the *Function Orchestration* category in the *technical* view of our classification framework. C denotes *custom DSL*, and O denotes *orchestrating function*, with the list of supported programming languages for developing orchestrating functions given in square braces. The abbreviations "n/s" and "n/a" stay for "not specified" and "not applicable", respectively.

	Orchestrator	Workflow definition	Control flow described	Quotas
Apache Openwhisk	Openwhisk composer	O [JavaScript, Python]	✓	Execution time
AWS Lambda	AWS step functions	C	✓	Execution time, I/O size
Fission	Fission workflows	C	✓	n/s
Fn	Fn Flow	O [Java]	✓	n/s
Google Cloud Functions	n/s	n/a	n/a	n/a
Knative	Knative eventing	C	✓[a]	n/s
Kubeless	n/s	n/a	n/a	n/a
MS Azure Functions	Azure durable functions	O [C#, JavaScript]	✓	n/s
Nuclio	n/s	n/a	n/a	n/a
OpenFaaS	n/s	n/a	n/a	n/a

[a]Only sequence and parallel execution are supported.

3.2.5 Testing e debugging

La maggior pare delle piattaforme supporta testing funzionale e debugging delle funzioni. La differenza sta nella sofisticatezza di questi sistemi, che è più elevata per le piattaforme commerciali.

3.2.6 Logging

Le piattaforme commerciali usano strumenti ad hoc per monitorare lo stato del servizio mentre quelle open source si rifanno a servizi terzi.

	Monitoring	Logging	Tooling Integr.
Apache Openwhisk	Kamon, Prometheus, Datadog	Logback (slf4j)	n/s
AWS Lambda	AWS CloudWatch	AWS CloudTrail, CloudWatch	n/s
Fission	Istio + Prometheus	Fission Logger + InfluxDB, Istio + Jaeger	Using a service mesh
Fn	Prometheus, Zipkin, Jaeger	Docker container logs	Push-based
Google Cloud Functions	Google Cloud Operations	Google Cloud Operations	n/s
Knative	Prometheus + Grafana, Zipkin, Jaeger	ElasticSearch + Kibana, Google Cloud Operations	Push-based
Kubeless	Prometheus + Grafana	n/s	n/s
MS Azure Functions	Azure Application Insights	Azure Application Insights	n/s
Nuclio	Prometheus, Azure Application Insights	stdout, Azure Application Insights	Push-based, pull-based
OpenFaaS	OpenFaaS Gateway + Prometheus	Kubernetes cluster API, Swarm cluster API, Loki	Pull-based

3.2.7 Application delivery

La maggior parte delle piattaforme segue un approccio dichiarativo per automatizzare il deploy delle applicazioni. Solo quelle commerciali supportano la pipeline di tipo CI/CD (Continuous Implementation/Continuous Deployment).

Classification of considered FaaS platforms, based on the *Application Delivery* category in the *technical* view of our classification framework. P and T denote *Platform-native tooling* and *third party tooling*, respectively. The abbreviation "n/s" stays for "not specified", meaning that no related information is documented.

	Deployment automation	CI/CD
Apache Openwhisk	wskdeploy (P)	n/s
AWS Lambda	AWS Cloud Formation (P), AWS SAM (P)	AWS CodePipeline(P)
Fission	Fission CLI (P)	n/s
Fn	Fn CLI (P)	n/s
Google Cloud Functions	Google Cloud Deployment Manager	Cloud Build (P)
Knative	Kubernetes (P)[a]	n/s
Kubeless	Kubernetes (P)[a], Serverless Framework (T)	n/s
MS Azure Functions	Azure Resource Manager (P)	Azure Pipelines (P), Azure App Service (P), Jenkins (T)
Nuclio	nuctl (P)[a]	n/s
OpenFaaS	faas-cli (P)	OpenFaaS Cloud (P), faas-cli (P), Circle CI (T), CodeFresh (T), Drone CI (T), GitLab CI/CD (T), Jenkins (T), Travis (T)

[a]Using Kubernetes specification with Custom Resource Definitions.

3.2.8 Riuso

Solamente AWS Lambda e MS Azure Functions prevedono un marketplace per condividere e riutilizzare funzioni.

3.2.9 Gestione degli accessi

Le piattaforme commerciali supportano nativamente l'autenticazione e il controllo delle risorse. Quelle open source di solito sfruttano features del sistema operativo.

4 PaaS

L'approccio PaaS prevede che i fornitori esterni gestiscano sia hardware che software e che l'utente fornisca solamente i dati e l'applicativo.
I **vantaggi** sono:

- Ridotta gestione per l'utente
- Manutenzione automatica
- Load balancing, scaling e distribuzione più efficienti
- Più facilità nell'adottare nuove tecnologie

Gli **svantaggi** invece:

- Disponibilità del servizio molto dipendente dal fornitore
- Vendor lock-in
- In balia di eventuali cambiamenti da parte del fornitore

4.1 Heroku

Heroku è una piattaforma cloud basata sulla gestione di un sistema di container, con data services integrati e un ampio ecosistema, per sviluppare ed eseguire app moderne.
Gli utenti usano *container* chiamati **dynos** per lanciare ed eventualmente scalare le loro applicazioni. Questi sono container linux virtualizzati ed isolati progettati per eseguire codice. Ne esistono di diversi tipi (e costi) a seconda delle necessità per l'applicazione.

4.1.1 Deployment

Per fare il deploy di un'applicazione, Heroku necessita di:

- Il **codice sorgente**
- Una lista di **dipendenze**
- Un **procfile**, ovvero un file contenente l'elenco dei comandi necessari a far partire il codice

Il sistema, una volta ricevuti questi parametri, ottiene i linguaggi e le dipendenze necessarie e produce uno **slug**, che poi verrà inviato ad un *dyno*.

4.1.2 Runtime

Quando l'applicazione viene fatta partire, Heroku crea un numero di *dyno* in base al carico previsto, ognuno caricato con la stessa configurazione fornita dall'utente.

4.1.3 Add-ons

Esistono più di 150 servizi di terze parti che forniscono add-ons per le applicazioni che hanno funzionalità di molti tipi diversi, tra cui monitoraggio, data store e logging.
Questo porta al fenomeno del vendor lock in, in quanto gli add-on potrebbero non essere disponibili al di fuori di Heroku e potrebbe quindi essere necessario riscrivere il codice per quella funzionalità.

4.2 Altri fornitori

Altri esempi di PaaS sono:

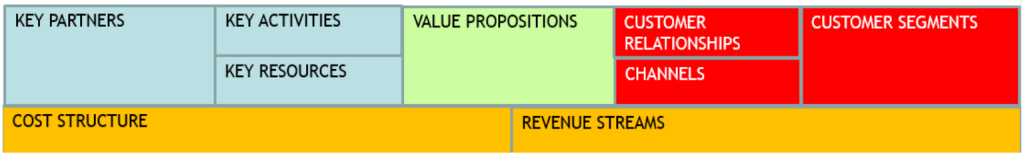
- Microsoft Azure
- Open shift
- Google Firebase

5 Business models

Il **business model** descrive l'idea di come un'organizzazione crea, consegna e cattura del **valore**.

5.1 Canvas

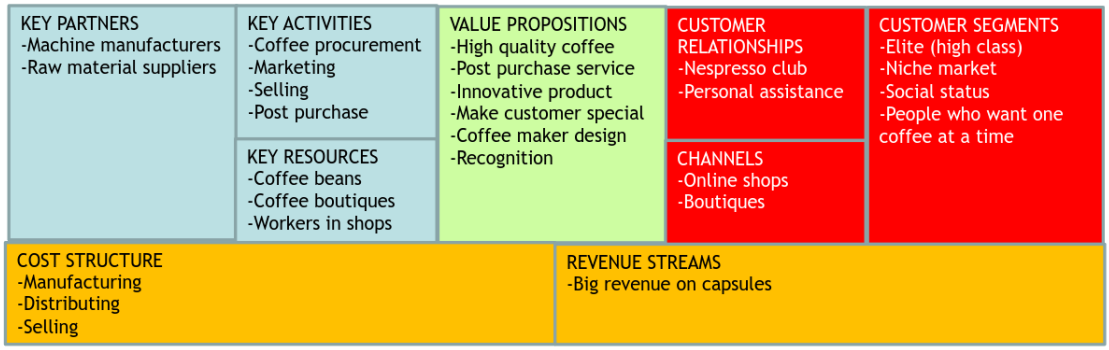
Il Business Model Canvas è un linguaggio condiviso per descrivere, visualizzare, valutare e modificare modelli di business. Si struttura come segue:



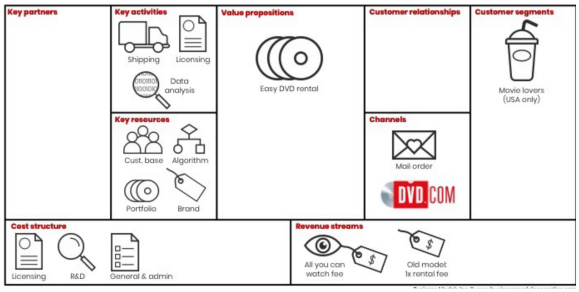
Vediamo nel dettaglio ogni sezione:

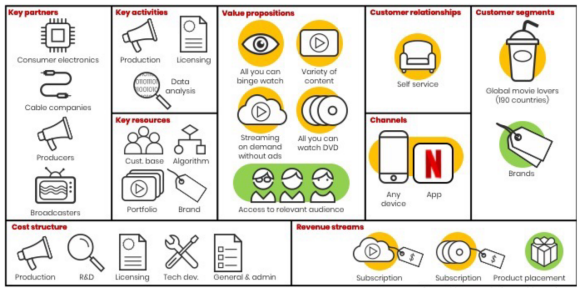
- **Value propositions**: descrive l'insieme di prodotti e servizi che creano valore per un segmento specifico di clienti
- **Customer**
  - **Segments**: definisce i diversi gruppi di persone o organizzazioni che un'impresa vuole raggiungere e servire
  - **Relationship**: descrive i tipi di relazioni che un'azienda stabilisce con determinati segmenti
  - **Channels**: descrive come l'azienda comunica e raggiunge i segmenti
- **Key**
  - **Resources**: le risorse più importanti per far funzionare un business model
  - **Activities**: le attività più importanti per far funzionare un business model
  - **Partners**: la rete di fornitori e partner che fanno funzionare un business model
- **Revenue streams**: il guadagno di un'azienda diviso per ogni segmento
- **Cost structure**: tutti i costi strutturali previsti dal business model

Esempio 5.1.1 (Nespresso)**.** Nel 1976 la *Nestré* dominava il mercato con il caffè istantaneo tramite il prodotto Nescafé.
Nel 1988 la Nespresso riesce a ribaltare la situazione concentrandosi su segmenti di mercato di elite. In particolare, usa il seguente canvas:



Esempio 5.1.2 (Netflix)**.** Netflix è riuscito a spodestare il dominio di aziende come Blockbuster introducendo lo streaming. Vediamo il confronto tra i due canvas:

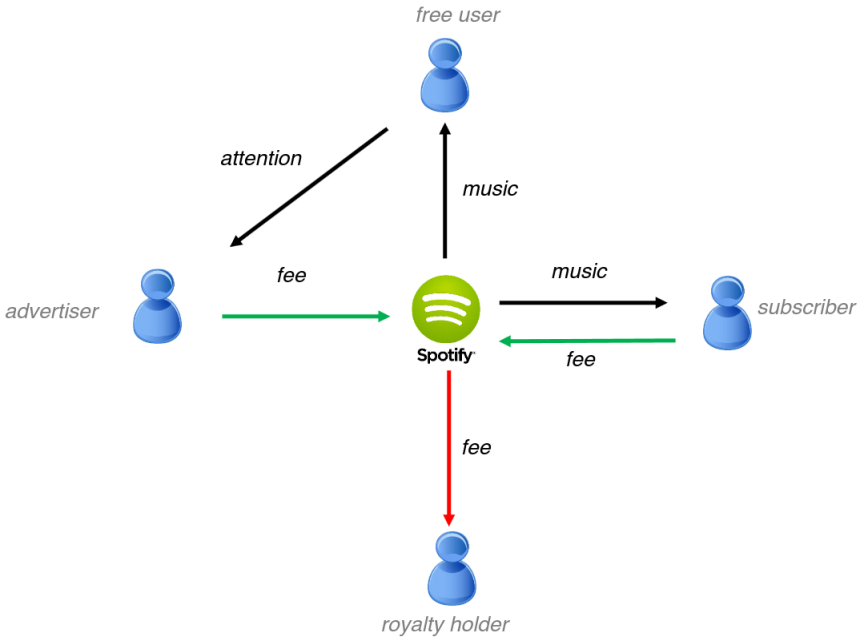




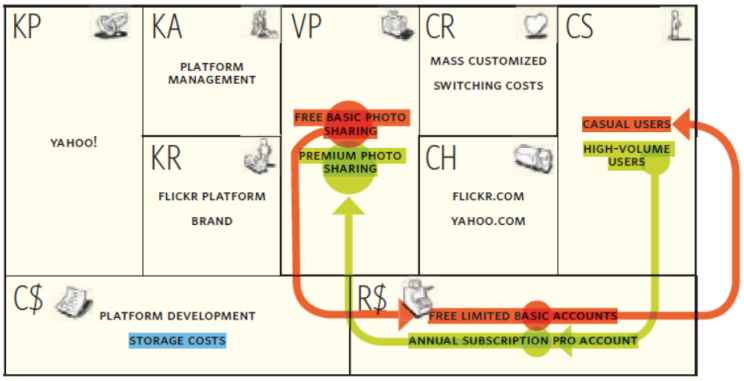
5.2 Freemium

L'idea è di combinare funzionalità di base gratuite con servizi più avanzati a pagamento, tenendo d'occhio quanto costa all'azienda mantenere i servizi gratuiti e a che rateo gli utenti diventano premium. In media il 10% di tutti gli utenti diventano premium.

Esempio 5.2.1 (Spotify)**.** Spotify ha gli utenti gratuiti che fanno guadagnare tramite le pubblicità e quelli a pagamento che pagano l'abbonamento. Spotify deve poi pagare le commissioni a chi detiene i diritti sulla musica.

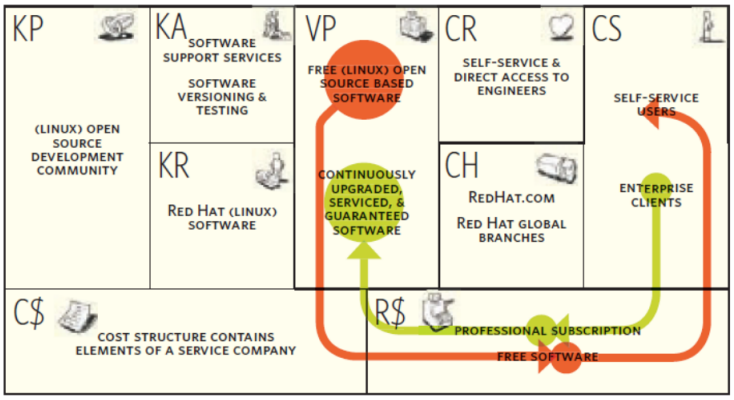


Esempio 5.2.2 (Flickr)**.** Vediamo il canvas di Flickr:

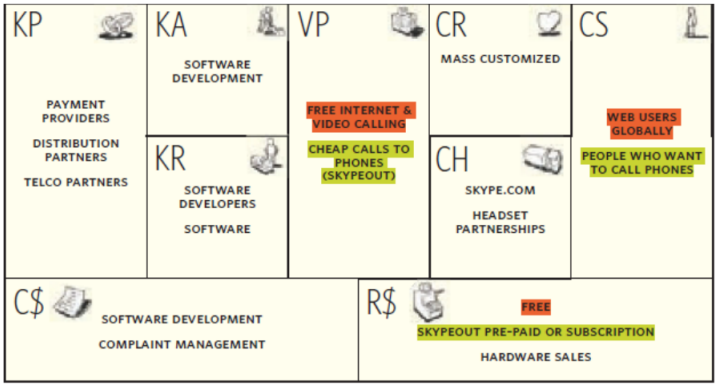


Anche Flickr si basa su un principio *freemium*: in rosso il tier gratuito mentre in verde quello a pagamento.

Esempio 5.2.3 (Redhat)**.** Redhat si concentra sul segmento di mercato delle aziende che vogliono sfruttare prodotti opensource ma che vogliono anche che ci sia qualcuno di responsabile (anche legalmente).



Esempio 5.2.4 (Skype)**.** Permette di fare chiamate gratuite attraverso internet.



Note 5.2.1. Anche Dropbox utilizza un principio freemium. A differenza di Netflix, Dropbox si è poi distaccato da Amazon AWS e ha creato la sua infrastruttura.

5.3 Innovazione

L'innovazione parte dal cambio di prospettiva: bisogna concentrarsi su quella del cliente. Da qui poi possiamo farci le domande **what if...?**, che ci portano ad identificare principalmente quattro epicentri di miglioramento:

- **Resource Driven**: queste innovazioni nascono da un'infrastruttura preesistente per *espandere* o *trasformare* il business model. Un esempio è Amazon AWS.
- **Customer Driven**: queste innovazioni si basano sulle *necessità del cliente*, sulla *facilità di accesso* o sul migliorare la *comodità*. Influenza delle parti specifiche del business model. Un esempio è 23andMe che offre test del DNA specifici per certe richieste.
- **Offer Driven**: le innovazioni di questo tipo creano una nuova *value proposition*, ad esempio Cemex consegna il cemento in 4 ore invece che 48.
- **Finance Driven**: innovazioni che *riducono i costi* o *aumentano i guadagni* (anche introducendo nuove fonti). Un esempio è Xerox che fornisce le prime 2000 copie gratuite.

5.4 Casi di studio

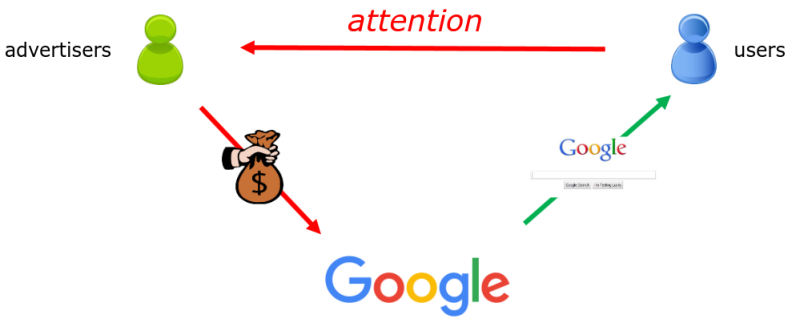
5.4.1 Amazon

Amazon (originariamente *Cadabra*) viene fondata da Jeff Bezos nel 1994 come negozio online di libri. Non si aspettava di avere profitti per i primi 4 o 5 anni e infatti il primo quadrimestre positivo è stato l'ultimo del 2001.
Al 2022 i guadagni (in miliardi di dollari) si dividono in:

1. 231.87 dallo store online
2. 140.05 dai servizi ai venditori di terze parti
3. 90.76 da AWS
4. 40.20 da abbonamenti (e.g. Prime)
5. 20.03 da negozi fisici

5.4.2 Google

I guadagni di Google si concentrano sulle **pubblicità personalizzate**:



non solo tramite il motore di ricerca ma anche con tutte le altre applicazioni che forniscono.

6 Software services

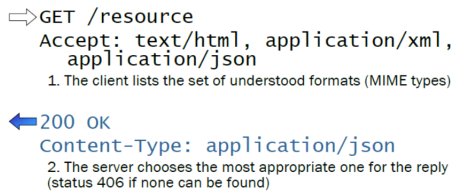
6.1 REST

REST (REpresentational State Trasfer) è uno stile architetturale sviluppato come modello astratto per il Web.
Ogni azione causa il passaggio dell'applicazione allo stato successivo tramite il trasferimento dello stato corrente.
Si basa sui seguenti principi:

- **URIs (Uniform Resource Identifier)**: identificazione delle risorse in maniera *univoca* e *universale*, ad esempio gli indirizzi Web
- **Interfaccia uniforme**: ad esempio per il protocollo HTTP, l'utente invoca dei metodi predefiniti:
  - *POST* e *PUT* per creare e aggiornare risorse
  - *DELETE* per eliminare una risorsa
  - *GET* per ottenere lo stato corrente di una risorsa
- **Messaggi autoesplicativi**: ogni richiesta contiene abbastanza *contesto* per riuscire a comprendere il messaggio. Inoltre le risorse sono *separate* dalla loro rappresentazione in modo da poter essere accedute in diversi formati
- **Interazioni stateful tramite hyperlinks**: data la natura stateless, per fornire interazioni stateful si utilizzano trasferimenti espliciti di stato tramite hyperlinks

6.1.1 Negoziazione

Quando viene richiesta una risorsa, vanno specificati i formati accettabili. Il server poi risponde con la risorsa nel formato più appropriato o eventualmente con il codice di errore 406.



6.1.2 Design

Il design deve seguire i seguenti passaggi:

1. Identificare le risorse che devono essere esposte come servizi
2. Modellare le relazioni tra le risorse tramite gli *hyperlinks*
3. Definire gli URIs seguendo le seguenti guidelines:
   - Meglio i sostantivi dei verbi
   - Brevità
   - Utilizzare i template per costruire ed elaborare URIs parametrici
   - Non cambiarli, nel caso utilizzare il reindirizzamento
4. Definire quali metodi sono applicabili alla risorsa
5. Progettare e documentare la rappresentazione delle risorse
6. Implementare e rilasciare il servizio
7. Testing

6.1.3 Riepilogo

L'architettura REST è **semplice** (utilizza standard famosi e infrastruttura già presente), **leggera** (sia per i protocolli utilizzati che per i messaggi) e **scalabile**. Di contro però i client possono richiedere **troppi o troppi pochi dati**, possono fare un **numero limitato di richieste** e le convenzioni per la **nomenclatura** sono inconsistenti.

6.2 OpenAPI

L'iniziativa OpenAPI (ideata dalla Linux Fundation Collaborative Project) ha come obiettivo quello di creare uno standard neutro ed indipendente di API di tipo REST. Formalmente chiamata **Swagger**, si basa su una descrizione in JSON degli endpoint HTTP, come vengono usati e la struttura dei loro parametri in input e output.
In questo modo l'applicazione può richiedere le specifiche della API per poi fare le richieste nel formato corretto.

6.3 Microservizi

I microservizi nascono per due motivi principali:

- Diminuire il **tempo** necessario ad aggiungere funzionalità o aggiornamenti (si riduce il tempo di build)
- Aumentare la **scalabilità**

In particolare, l'applicazione viene vista come insieme di servizi, ognuno in esecuzione sul proprio container **indipendente** che comunica con gli altri tramite meccanismi leggeri di comunicazione. Questo garantisce che la gestione dei dati sia **decentralizzata** e che l'applicazione sia **scalabile** orizzontalmente. Garantisce infine una buona **tolleranza ai guasti**.
A livello di team, generalmente ce n'è uno per servizio che si occupa dell'intero lifecycle, dallo sviluppo al deploy.
I due lati negativi principali sono:

- **Overhead di comunicazione**: dato che tutti i servizi devono comunicare tra di loro, ci sarà un grosso traffico
- **Complessità** del sistema

7 Kubernetes

Kubernetes è un **orchestratore** di container. In pratica si assicura che gruppi di container che devono lavorare assieme lo facciano in maniera corretta ed efficiente (un po' come il direttore d'orchestra dà le istruzioni ai musicisti).
Si occupa dell'intero **ciclo di vita** del container, dall'*allocazione* di risorse allo *spegnimento*, nonché della *comunicazione* tra container e della *schedule* ottimale per completare un lavoro.

7.1 Funzionamento

La struttura di un sistema con Kubernetes si compone di un **orchestratore** che espone delle **API** e che è connesso con molteplici **worker**, ognuno dei quali è un container a sé stante con un **kubelet**, ovvero un processo che si occupa di comunicare con l'esterno l'orchestratore.
All'orchestratore viene data in pasto una configurazione che rappresenta lo **stato desiderato**, e lui poi farà in modo, con le risorse che possiede, di mantenerla sempre attiva. Questa configurazione contiene principalmente:

- **Deployment**
- **Pod**: unità più piccola che può contenere una o più immagini di container e della quale si deve specificare il numero di istanze necessarie

Nel momento in cui un worker smette di funzionare, l'orchestratore sposta i pod che vi erano in esecuzione su un altro worker, in modo da mantenere costante il numero di repliche attive.

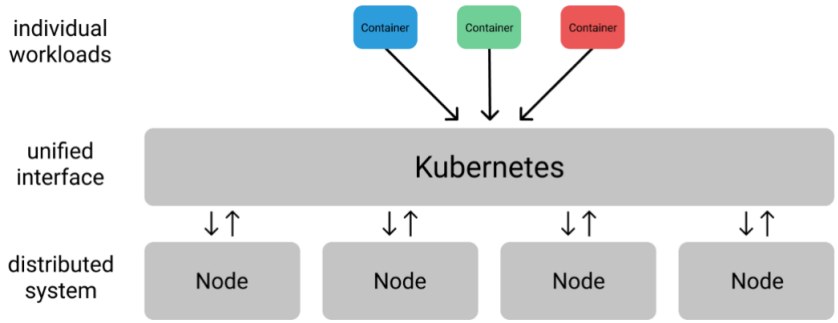
7.2 Principi

7.2.1 Dichiaratività

Dichiariamo lo **stato desiderato** del sistema e lasciamo che Kubernetes si occupi di fare tutto ciò che è necessario per raggiungere quell'obiettivo e risolvere eventuali problemi che possono presentarsi.
Lo stato desiderato lo definiamo tramite degli **oggetti**, ognuno con delle specifiche che forniscono lo stato desiderato e lo stato attuale. Kubernetes si occupa poi di verificare costantemente che lo stato attuale sia equivalente a quello desiderato e, in caso non lo sia, provare a ripararlo o sostituirlo direttamente con uno nuovo.

7.2.2 Distribuzione

Kubernetes fornisce un'interfaccia unificata per interagire con un cluster di macchine, evitandoci di doverci preoccupare della comunicazione con le singole.



7.2.3 Decoupling

I container devono essere sviluppati in modo che abbiano un singolo compito da svolgere, seguendo i principi dei **microservizi**.

7.2.4 Infrastruttura immutabile

Per ottenere i maggiori benefici dall'orchestrazione di container dobbiamo lavorare basandoci sul principio di un'infrastruttura immutabile. Ad esempio invece di aggiornare eventuali librerie di un container, distruggerlo e ricrearlo da zero con gli aggiornamenti già pronti.
Di conseguenza durante l'intero ciclo di vita di un container dobbiamo utilizzare la stessa configurazione ed eventualmente sostituirli con nuovi.
Tutto questo ci permette di rendere molto semplice il rollback ad una vecchia versione.

7.3 Oggetti

Esistono molti oggetti in Kubernetes, noi ne vedremo solo i principali.

7.3.1 Manifesto

Il **manifesto** contiene la definizione degli oggetti di Kubernetes ed è scritto in *YAML* o *JSON*. Ogni oggetto ha le seguenti informazioni:

- Che **API** usa e quale versione
- Che **tipo** di oggetto è
- Cosa **identifica** in maniera univoca l'oggetto
- Lo **stato** che deve avere l'oggetto

7.3.2 Pod

Un **pod** consiste in un insieme di uno o più *container*, un livello **network** condiviso e volumi del **filesystem** condivisi.

7.3.3 Deployment

Un oggetto di tipo **deployment** contiene una raccolta di *Pod* definiti da un template e il numero di **repliche** necessarie per ognuno di essi. Il cluster cercherà in ogni modo di mantenere attive le n repliche indicate del template.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
   name: ml-model-serving
   labels:
      app: ml-model
spec:
   replicas: 10
   selector:
      matchLabels:
         app: ml-model
   template:
      metadata:
         labels:
            app: ml-model
      spec:
         containers:
         - name: ml-rest-server
            image: ml-serving:1.0
            ports:
            - containerPort: 80
```

7.3.4 Service

Ogni *pod* ha un indirizzo IP assegnato che viene usato per la comunicazione. Data la volatilità dei pod, che possono cessare di esistere o essere sostituiti velocemente, l'oggetto **service** fa il lavoro sporco del capire chi contattare, esponendo all'utente solo degli *endpoint*. Per farlo si basa sulle **labels**.

```
apiVersion: v1
kind: Service
metadata:
   name: ml-model-svc
   labels:
      app: ml-model
spec:
   type: ClusterIP
   selector:
      app: ml-model
   ports:
   -  protocol: TCP
      port: 80
```

7.3.5 Ingress

Se *Service* permette la comunicazione tramite un unico endpoint nella traffico locale, **ingress** permette la comunicazione con l'esterno.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
   name: ml-product-ingress
   annotations:
      kubernetes.io/ingress.class: "nginx"
      nginx.ingress.kubernetes.io/rewrite-target: /
spec:
   rules:
   -  http:
         paths:
         -  path: /app
            backend:
               serviceName: user-interface-svc
               servicePort: 80
```

7.4 Control plane

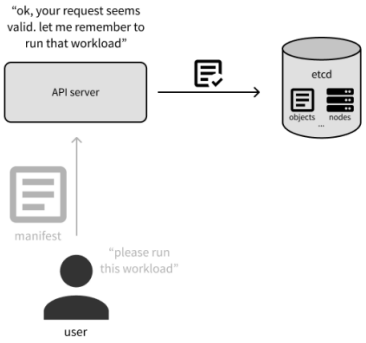
In un sistema di Kubernetes abbiamo due tipi di macchine:

- **Master node**: spesso singola, contiene la maggior parte dei componenti del pannello di controllo
- **Worker node**: macchina che esegue i workflow dell'applicazione

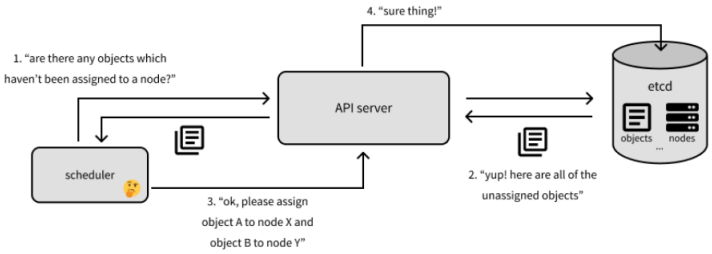
7.4.1 Master node

Si compone di:

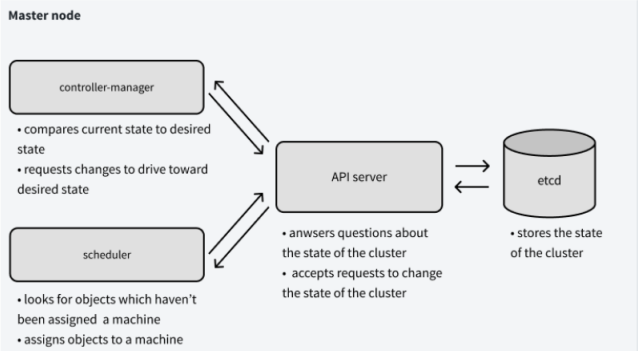
- Il **server API** convalida la richiesta e fornisce le informazioni sullo stato del cluster, che è salvato in uno storage distribuito di tipo chiave-valore chiamato **ectd**.
  Lo stato contiene diverse informazioni tra cui la configurazione attuale, le specifiche degli oggetti, i loro stati, i nodi presenti e che lavori hanno assegnati.



- Lo **scheduler** si occupa di decidere su che nodo far partire un determinato workload richiesto:
  1. Chiede al *server API* quali oggetti non sono stati assegnati a delle macchine
  2. Determina a quali macchine dovrebbero essere assegnati
  3. Risponde al *server API* comunicandogli gli assegnamenti eseguiti



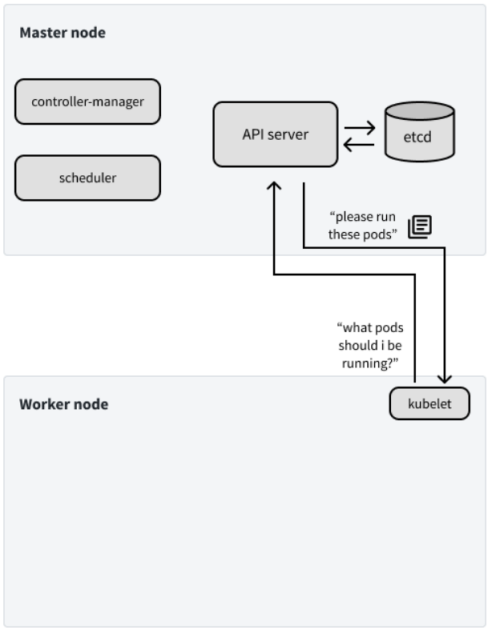
- Il **controller manager** monitora lo stato del cluster attraverso il *server API*. Se lo stato attuale non è quello desiderato, esegue richieste al server in modo da farlo convergere a quello voluto



7.4.2 Worker node

Il worker node si compone di due elementi:

- **kubelet**: è la parte del nodo che comunica con il *server API*. È responsabile dell'attivazione dei *pod* necessari al workload da eseguire. Appena un nodo entra nel cluster, il *kubelet* lo annuncia al *server API* in modo che lo scheduler possa assegnarli dei workload.



- **kube-proxy**: permette ai container di comunicare tra di loro attraverso i nodi del cluster.

7.5 Conclusione

Kubernetes non va usato quando:

- Il lavoro è leggero e può essere eseguito su una sola macchina
- Non è necessario che il servizio sia sempre disponibile
- Non sono previste molte modifiche
- È progettato in maniera *monolitica*

In questi casi è infatti più comodo e veloce usare **Docker swarm**.