

Corso di Laurea in Ingegneria e Scienze Informatiche

Prompt-to-Action: un Model Context Protocol per l'integrazione real-time con configuratori 3D

Tesi di laurea in:
COMPUTER GRAPHICS

Relatore
Prof. Damiana Lazzaro

Candidato
Latini Luca

Correlatore
Dott. Christian Lillini

Sessione di laurea unica Sessione di Laurea
Anno Accademico 2024-2025

Sommario

Il presente lavoro di tesi descrive la progettazione e realizzazione di un prototipo di Model Context Protocol (MCP) volto a consentire il controllo e la configurazione di un configuratore grafico 3D mediante l'uso di prompt testuali. L'obiettivo principale è stato definire e implementare un meccanismo che traduca comandi descritti in linguaggio naturale nella chiamata automatica degli strumenti appropriati e nell'esecuzione delle operazioni corrispondenti all'interno del configuratore. Il sistema è realizzato in C# e si articola in moduli per la gestione del contesto, l'invocazione dei tool e la comunicazione in tempo reale con il client grafico tramite WebSocket. Il lavoro comprende l'analisi dei requisiti, la progettazione dell'architettura software, l'implementazione dei moduli principali e la validazione funzionale tramite scenari di test. I risultati dimostrano la fattibilità del paradigma prompt→tool per operazioni di gestione progetto nel configuratore, evidenziando punti di forza e limiti attuali in termini di robustezza semantica, gestione degli errori e scalabilità. Come contributo si propone un prototipo funzionante e linee guida per future estensioni, quali la gestione multi-utente.

Optional. Max a few lines.

Indice

Sommario	iii
Introduzione	1
1 Fondamenti Teorici e Contesto Tecnologico	5
1.1 La Crisi della Frammentazione nell'Ecosistema AI	6
1.2 Definizione e Ruolo del Model Context Protocol (MCP)	6
1.2.1 Architettura e Componenti Centrali	7
1.2.2 Primitive Bidirezionali e Interazione Utente	10
1.2.3 Vantaggi Sistemici dell'Adozione MCP	13
1.3 WebSocket: Protocollo per Comunicazione Real-Time	14
1.3.1 Caratteristiche architettoniche	14
1.3.2 Vantaggi operativi e prestazionali	15
1.3.3 Limitazioni intrinseche e sfide implementative	16
1.3.4 Adozione di SignalR nel contesto progettuale	17
1.4 Panoramica e contesto su i Configuratori 3D	18
1.4.1 Gestione delle Regole e Motore di Configurazione	18
1.4.2 Calcolo Dinamico e Output del Processo	19
1.4.3 Architettura del Sistema: Frontend, Backend e Scambio Dati	20
1.4.4 Casi d'uso e applicazioni dei configuratori 3D	21
1.4.5 Sfide dell'Integrazione AI-3D	24
2 Architettura Tecnologica del Progetto	27
2.1 Panoramica dello Stack	28
2.2 Piattaforma Backend .NET	30
2.2.1 .NET 8 e C# 12: Piattaforma e Linguaggio	30
2.2.2 ASP.NET Core: Infrastructure Framework	32
2.2.3 MCP SDK: Astrazione Tool Definition	33
2.3 Architettura della Comunicazione	35
2.3.1 Protocolli di Base: REST, JSON-RPC, WebSocket	35
2.3.2 SignalR: Framework Real-Time	37

INDICE

2.4	Sicurezza Applicativa	40
2.4.1	JWT: Autenticazione Stateless	40
2.4.2	CORS e Transport Security	42
2.5	Integrazione Frontend	46
2.5.1	TypeScript: Type-Safety JavaScript	46
2.5.2	SignalR Client Browser	47
3	Progettazione del Sistema	51
3.1	Architettura Generale del Sistema	51
3.2	Sistema di Configurazione e Bootstrap	54
3.3	Servizio di Comunicazione Real-Time	57
3.4	Infrastruttura SignalR e WebSocket	60
3.4.1	Architettura Multi-Layer del Bridge	60
3.4.2	Modello delle Classi e Pattern Hub	61
3.4.3	Gestione del Ciclo di Vita delle Connessioni	64
3.4.4	Pattern di Comunicazione: Broadcasting e Point-to-Point . .	69
3.4.5	Flusso di Elaborazione Messaggi	72
3.4.6	Integrazione con l'Architettura MCP	75
3.5	Sistema di Autenticazione	77
3.5.1	Strategia Token-Based	77
3.5.2	Flusso di Autenticazione e Auto-Login	78
3.6	Gestione Progetti	79
3.7	Gestione Articoli e Varianti	80
3.8	Protocollo di Messaggistica SignalR	83
3.8.1	Semantica dei Messaggi e Command Pattern	84
3.8.2	Orchestrazione dei Comandi Server-Side	84
3.9	Flussi di Lavoro Principali	87
3.9.1	Caso d'Uso: Login e Recupero Progetti	87
3.9.2	Caso d'Uso: Creazione e Apertura Progetto	89
3.9.3	Caso d'Uso: Aggiunta Articolo con Varianti	91
3.9.4	Caso d'Uso: Comunicazione Bidirezionale	92
3.9.5	Caso d'Uso: Gestione Errori	92
4	Implementazione	97
4.1	Ambiente di Sviluppo e Setup Progetto	97
4.2	Implementazione MCP Server Core	97
4.2.1	Bootstrap e Dependency Injection	97
4.2.2	Implementazione Tool MCP	97
4.2.3	Implementazione Tool Complessi	97
4.3	Implementazione SignalR Service	97
4.3.1	Classe SignalRService e Connection Management	97

INDICE

4.3.2	Invio e Ricezione Messaggi	97
4.4	Implementazione Hub (SignalR Server)	97
4.5	Implementazione Client Configurator3D	97
4.6	Gestione Configurazione e Sicurezza	97
5	Conclusioni e sviluppi futuri	99
		101
	Bibliografia	101

INDICE

Elenco delle figure

1.1	Architettura del primitive Sampling nel cro:mcp <i>Model Context Protocol (MCP)</i> che mostra il ciclo di richiesta-inferenza-approvazione tra client, server e utente umano.[Mod25b]	11
1.2	Architettura del primitive Elicitation nell'MCP che mostra il ciclo di richiesta-interazione umana-risposta tra Server, Client e utente.[Mod25b]	12
2.1	Diagramma di sequenza del flusso CORS Preflight tra Frontend e Backend. Il server autorizza esplicitamente l'origine e l'uso delle credenziali prima della connessione WebSocket.	44
3.1	Diagramma 1: Architettura di alto livello: componenti principali e relazioni tra Program, ExternalApiTools, SignalRService, ExternalApi e Configurator3D.	52
3.2	Diagramma 2: Layer di configurazione: struttura di AppSettings (con ExternalCredentials e SignalRConfig) e relazioni con Program, IServiceCollection e IConfiguration.	55
3.3	Diagramma delle classi del SignalR Service Layer: astrazione della connessione e interfacce verso l'Hub.	58
3.4	Diagramma delle classi del livello di comunicazione: struttura del SignalRHub, definizione del contratto IClient e composizione del DTO Message.	62
3.5	Diagramma di sequenza del flusso di connessione e messaggistica tra Configurator3D, IwineHub e framework SignalR.	66
3.6	Macchina a stati della connessione SignalR: fasi di negoziazione, operatività e disconnessione gestite dal framework.	68
3.7	Flusso di Broadcasting: propagazione parallela del messaggio (soluzione adottata nel prototipo).	71
3.8	Pipeline logica di elaborazione del messaggio: dalla serializzazione all'esecuzione sul client.	74
3.9	Diagramma delle classi per la gestione articoli: interazione tra i tool MCP e le primitive del gestionale esterno.	82

ELENCO DELLE FIGURE

3.10 Diagramma di sequenza del flusso di autenticazione e recupero progetti: gestione intelligente del token e delle credenziali	87
3.11 Diagramma di sequenza del flusso di creazione e apertura progetto: orchestrazione tra API REST e canale SignalR.	89
3.12 Diagramma di sequenza del flusso di aggiunta di un articolo con una variante.	91
3.13 Gestione degli errori infrastrutturali: scadenza token e indisponibilità del servizio SignalR.	93
3.14 Gestione degli errori logici: entità non trovate e fallimenti parziali nella configurazione.	94

List of Listings

LIST OF LISTINGS

Introduzione

Il lavoro di ricerca e sviluppo descritto in questa tesi è stato condotto nell'ambito di un tirocinio presso **Apra S.p.A.**, una realtà consolidata nel panorama delle software house italiane, attiva da oltre quarant'anni nella trasformazione digitale delle imprese. L'azienda si distingue per la sua competenza in settori strategici quali Cloud Computing, Big Data, Internet of Things e Industria 4.0, offrendo consulenza e soluzioni tecnologiche mirate all'ottimizzazione dei processi di business.

L'attività è stata svolta all'interno del team di Ricerca e Sviluppo, un'unità focalizzata sull'esplorazione di nuove paradigmi per l'orchestrazione tra servizi di Intelligenza Artificiale e applicazioni enterprise. In questo contesto, è emerso un forte interesse verso l'integrazione tra il Model Context Protocol (MCP) e interfacce utente specializzate, nello specifico configuratori grafici tridimensionali. Tale ambito, sebbene ancora poco esplorato in letteratura, presenta un potenziale significativo per l'evoluzione dei flussi di lavoro aziendali, promettendo di semplificare l'interazione uomo-macchina in scenari complessi.

Il progetto ha avuto una natura prettamente esplorativa, con l'intento di verificare la fattibilità tecnica di un sistema in cui un server MCP possa controllare un ambiente 3D. L'indagine si è concentrata sulla prototipazione e sulla valutazione dei pattern di integrazione necessari per far dialogare il server MCP con il client grafico attraverso un bridge *real-time*. L'attenzione è stata posta sui vincoli operativi critici, quali i meccanismi di autenticazione, la sincronizzazione istantanea degli stati e la robustezza semantica nell'interpretazione dei comandi. Il risultato atteso non consiste in un prodotto commerciale finito, bensì in un insieme di evidenze empiriche e raccomandazioni architetturali che possano guidare futuri sviluppi, come l'eventuale integrazione di moduli di Natural Language Processing (NLP) o il supporto multi-utente.

Dal punto di vista infrastrutturale, il progetto ha preso le mosse da due asset tecnologici già presenti in azienda: un sistema di comunicazione *real-time* basato su SignalR e un configuratore grafico 3D, responsabile del rendering e della logica di visualizzazione. Il compito fondamentale è stato quello di orchestrare questi elementi, sviluppando ex novo il server MCP e adattando il client grafico affinché potesse stabilire una connessione bidirezionale, interpretare i messaggi in ingresso e tradurli in azioni concrete sulla scena tridimensionale.

L'obiettivo generale del lavoro è la progettazione e la realizzazione di un prototipo operativo basato sul Model Context Protocol, capace di pilotare un configuratore 3D attraverso prompt testuali. Tale implementazione mira a validare la fattibilità dell'integrazione *real-time* e a tracciare una roadmap tecnica per una possibile industrializzazione della soluzione.

Nello specifico, il progetto si è articolato in diverse fasi implementative. In primo luogo, è stato necessario sviluppare il nucleo logico del server MCP in linguaggio C#, definendo i meccanismi che mappano le richieste testuali (prompt) in strumenti e comandi eseguibili (tool). Parallelamente, l'integrazione ha richiesto lo sfruttamento del bridge esistente basato su SignalR e WebSocket per stabilire un canale di comunicazione bidirezionale stabile e reattivo tra il server e l'interfaccia grafica.

Un aspetto cruciale ha riguardato l'adattamento del configuratore 3D: il software client è stato esteso per gestire la connessione al canale di messaggistica, decodificare le istruzioni provenienti dal server MCP e invocare le funzioni interne necessarie per aggiornare la vista o modificare il modello in tempo reale. A supporto dell'interazione, si è lavorato anche sul front-end, sviluppato in Svelte, per garantire un'interfaccia utente efficace nell'invio dei prompt e nella visualizzazione del feedback di sistema.

La validazione del sistema è stata condotta attraverso la definizione e la verifica di molteplici scenari d'uso *end-to-end*, che spaziano dalla creazione di un nuovo progetto all'aggiunta di componenti specifici, fino alla consultazione del catalogo prodotti, coprendo le funzionalità core tipiche di un flusso di configurazione.

È opportuno precisare che il perimetro del progetto si è limitato all'integrazione e all'adattamento dei sistemi, escludendo lo sviluppo da zero di motori *real-time* proprietari o la riscrittura del motore grafico. Inoltre, l'interpretazione dei comandi

si basa su strutture predefinite, rimandando a sviluppi futuri l'implementazione di algoritmi avanzati per la comprensione del linguaggio naturale libero.

Il presente lavoro si articola in sei capitoli, strutturati per accompagnare il lettore dall'analisi teorica alla realizzazione pratica del sistema:

- Il **Capitolo 1 (Introduzione)** presenta il contesto aziendale in cui è maturato il progetto presso Apra S.p.A., definendone le motivazioni e gli obiettivi principali.
- Il **Capitolo 2 (Background)** approfondisce lo stato dell'arte relativo al *MCP*, alla comunicazione *real-time* tramite WebSocket e ai principi di funzionamento dei moderni configuratori 3D.
- Il **Capitolo 3 (Stack Tecnologico)** descrive l'ecosistema software adottato, motivando la scelta di .NET 8, SignalR e delle tecnologie frontend impiegate.
- Il **Capitolo 4 (Progettazione del Sistema)** illustra l'architettura logica del prototipo, i moduli funzionali che lo compongono e i protocolli di scambio dati definiti per l'interazione tra IA e ambiente grafico.
- Il **Capitolo 5 (Implementazione)** analizza nel dettaglio lo sviluppo del server *mcp* in C#, l'integrazione del bridge di messaggistica e l'adattamento del client grafico per l'esecuzione dei comandi.
- Il **Capitolo 6 (Conclusioni e sviluppi futuri)** sintetizza i risultati raggiunti, valuta l'efficacia del paradigma prompt-to-action e delinea i possibili sviluppi per l'estensione delle capacità del sistema.

Capitolo 1

Fondamenti Teorici e Contesto Tecnologico

Questo capitolo introduce il contesto teorico e tecnologico di riferimento del lavoro di tesi. L'obiettivo è fornire le basi necessarie per comprendere le tecnologie utilizzate e le motivazioni che hanno guidato le scelte progettuali.

Nella prima parte viene analizzato il problema della frammentazione nell'ecosistema dell'intelligenza artificiale e viene presentato il Model Context Protocol come proposta di standard per migliorare l'interoperabilità tra modelli e sistemi esterni. Successivamente viene descritto il protocollo WebSocket, evidenziandone il ruolo nella comunicazione real-time e il suo utilizzo attraverso framework di più alto livello. L'ultima parte del capitolo è dedicata ai configuratori 3D, al loro funzionamento e alle principali sfide legate alla loro integrazione con sistemi di intelligenza artificiale.

Questo inquadramento permette di collocare il progetto all'interno di un contesto più ampio e di preparare il lettore alla descrizione delle soluzioni adottate nei capitoli successivi.

1.1 La Crisi della Frammentazione nell'Ecosistema AI

Modelli Linguistici di Grande Scala (LLM) sono diventati centrali nell'Intelligenza Artificiale (Artificial Intelligence) moderna, dimostrando capacità straordinarie nella comprensione e generazione del linguaggio naturale [ESGK25], e alimentando agenti autonomi che operano in ambienti cloud, edge e desktop[ESGK25]. Questi agenti sono cruciali per automatizzare compiti complessi ed eseguire azioni interagendo con servizi o strumenti esterni.[KD25]

Nonostante i rapidi progressi nel ragionamento degli LLM, essi rimangono intrinsecamente vincolati dalla dipendenza da dataset di addestramento statici, limitando la loro applicabilità in scenari dinamici e in tempo reale [SEKK25]. Tradizionalmente, l'integrazione degli LLM con sistemi esterni si è basata su interfacce di programmazione (*Application Programming Interface (API)*) frammentate e costruite su misura.[ESGK25]

Questa mancanza di standardizzazione crea una crisi di frammentazione[CSI⁺25], ostacolando la scalabilità, la sicurezza e la generalizzazione della comunicazione tra agenti guidati dagli LLM[ESGK25]. Le integrazioni ad-hoc comportano una duplicazione dello sforzo di sviluppo, aumentano la complessità, e introducono inconsistenze di sicurezza[SEKK25]. Per ottenere flussi di lavoro multi-agente modulari, riutilizzabili e resilienti, l'interoperabilità, la capacità dei sistemi distinti di scoprire capacità, scambiare contesto e coordinare azioni in modo fluido, è considerata essenziale.[ESGK25]

1.2 Definizione e Ruolo del Model Context Protocol (MCP)

Per rispondere a questa esigenza sistematica di standardizzazione, Anthropic ha introdotto il *MCP* nel novembre 2024 [Ant24]. Questo standard open-source nasce con l'ambizione di unificare il modo in cui le applicazioni basate su intelligenza artificiale comunicano con sistemi esterni, superando la frammentazione delle integrazioni ad-hoc che caratterizza l'attuale panorama tecnologico [Mod25e].

1.2. DEFINIZIONE E RUOLO DEL MODEL CONTEXT PROTOCOL (MCP)

La metafora utilizzata per descrivere l'MCP è quella della "porta USB-C per l'AI": così come lo standard USB-C ha reso universale la connettività fisica tra dispositivi eterogenei, l'MCP si propone di fornire un'interfaccia unificata attraverso cui gli LLM possono accedere a dati e strumenti esterni in modo predicibile e sicuro [Mod25e]. L'analogia non è casuale e si inserisce in una tradizione consolidata di protocolli di standardizzazione che hanno rivoluzionato i rispettivi domini: le *cro:api* *Application Programming Interface (API)* REST per l'interoperabilità web e il (*Language Server Protocol (LSP)*) per l'integrazione degli strumenti di sviluppo negli *cro:ide* *Integrated Development Environment (IDE)* moderni rappresentano precedenti significativi di questo approccio architetturale [Mod25e].

L'obiettivo fondamentale dell'MCP è permettere la sostituzione delle integrazioni frammentate e proprietarie con un protocollo condiviso che riduca gli sforzi in fase di sviluppo e migliorare l'efficacia degli LLM fornendo loro accesso strutturato a contesti dinamici e aggiornati, superando il limite intrinseco dei dataset di addestramento statici. Questo secondo aspetto è particolarmente rilevante in scenari applicativi dove la pertinenza delle risposte dipende dalla capacità del modello di accedere a informazioni in tempo reale, come cataloghi prodotti, database aziendali o interfacce di controllo di sistemi complessi.

1.2.1 Architettura e Componenti Centrali

L'architettura dell'MCP segue un modello client-server persistente progettato per gestire l'interazione strutturata tra modelli linguistici e risorse esterne. L'ecosistema è composto da tre organi principali ciascuno dei quali ha responsabilità ben definite che garantiscono la separazione dei compiti e la modularità del sistema [Mod25a].

L'MCP Host rappresenta l'applicazione di front-end che orchestra l'intera esperienza utente. Tipicamente rappresentato da strumenti come Claude Desktop o ambienti di sviluppo integrati, l'Host funge da contenitore per il modello linguistico e coordina le connessioni verso uno o più Server MCP. La sua responsabilità critica risiede nella gestione del consenso dell'utente. Prima di accedere a dati sensibili o eseguire azioni potenzialmente impattanti, l'Host deve richiedere e ottenere l'autorizzazione esplicita dell'utilizzatore finale. Inoltre, quando il contesto necessario per rispondere a una richiesta proviene da multiple sorgenti, l'Host

1.2. DEFINIZIONE E RUOLO DEL MODEL CONTEXT PROTOCOL (MCP)

si occupa di aggregare le informazioni raccolte dai vari client e di presentarle al modello in forma coerente [Mod25a].

L'**MCP Client** opera come componente di traduzione e mediazione. Ogni istanza di client mantiene una connessione dedicata uno-a-uno con un singolo Server MCP, responsabilizzandosi della conversione bidirezionale dei messaggi: trasforma le richieste dell'LLM, spesso espresse come chiamate a funzioni strutturate, nel formato wire del protocollo MCP, e viceversa converte le risposte del server in rappresentazioni comprensibili al modello linguistico. Il client gestisce anche la fase di discovery, identificando i server disponibili nell'ambiente e le capacità che ciascuno espone [Mod25a].

L'**MCP Server** costituisce il punto di integrazione effettivo con sistemi e dati esterni. Ogni server è progettato per concentrarsi su un dominio specifico, ad esempio l'accesso a un database, l'interazione con un'API REST, o il controllo di un'applicazione desktop. Questa specializzazione favorisce la riutilizzabilità e semplifica la manutenzione. I server possono operare in modalità locale, comunicando tramite standard input/output (Stdio) per minimizzare l'overhead di rete, oppure in modalità remota attraverso protocollli HTTP, permettendo deployment distribuiti e scenari cloud-native [Mod25a].

Dal punto di vista implementativo, il protocollo si articola su due strati ben distinti che riflettono la classica separazione tra semantica e trasporto nelle architetture di rete [SEKK25, Mod25a]. Il **Data Layer** definisce la struttura logica delle interazioni, basandosi sulla specifica JSON-RPC 2.0 per la serializzazione dei messaggi. Questo livello regola il ciclo di vita delle connessioni, inclusa la fase di negoziazione iniziale delle capacità supportate da client e server, e definisce i primitivi fondamentali del protocollo che verranno discussi nella sezione successiva. Il **Transport Layer**, invece, si occupa della trasmissione fisica dei dati. L'MCP supporta nativamente due modalità di trasporto: lo Stdio, ottimale per comunicazioni locali tra processi sulla stessa macchina grazie all'assenza di overhead di rete, e lo Streamable HTTP, che sfrutta richieste POST per i messaggi client-server e opzionalmente Server-Sent Events per lo streaming di risposte lunghe, con supporto integrato per meccanismi di autenticazione standard come bearer token o OAuth [Mod25a, ESGK25]. Il concetto di **primitive** rappresenta il nucleo semantico dell'MCP. Esse vanno a definire le modalità attraverso cui i server possono esporre

1.2. DEFINIZIONE E RUOLO DEL MODEL CONTEXT PROTOCOL (MCP)

funzionalità e contesto alle applicazioni basate su intelligenza artificiale. La specifica distingue tre categorie principali di primitive, ciascuna con un diverso modello di controllo che riflette la natura dell’interazione [Mod25c].

I **Tools** (strumenti) costituiscono primitive controllate dal modello linguistico stesso. Quando un LLM determina che per rispondere a una richiesta utente è necessario eseguire un’azione specifica, come interrogare un database o invocare un’API esterna, può decidere autonomamente di chiamare uno degli strumenti esposti dal server MCP. Questa capacità di invocazione dinamica rappresenta il cuore dell’agentic AI, permettendo ai modelli di andare oltre la semplice generazione testuale per interagire concretamente con sistemi esterni. Ad esempio, uno strumento potrebbe permettere al modello di creare un nuovo record in un gestionale aziendale o di recuperare informazioni in tempo reale da un catalogo prodotti [Mod25c].

Le **Resources** (risorse) sono invece primitive controllate dall’applicazione host. Rappresentano sorgenti di dati strutturati in sola lettura che arricchiscono il contesto disponibile al modello senza richiederne l’invocazione esplicita. Ogni risorsa è identificata da un uri univoco che ne specifica la natura e la locazione, seguendo convenzioni familiari come `file:///path/to/document.md` per file locali o schemi custom per risorse specializzate. Questo approccio dichiarativo permette all’applicazione di decidere quali contesti rendere disponibili, mantenendo il controllo sulla sicurezza e sull’accesso ai dati sensibili [Mod25c].

I **Prompts** (template) sono primitive controllate dall’utente finale, progettate per standardizzare e parametrizzare pattern di interazione ricorrenti. Attraverso template riutilizzabili, gli utenti possono definire flussi di lavoro complessi che combinano più operazioni in sequenze coerenti, garantendo consistenza nelle modalità di interrogazione del sistema. Questa categoria di primitive risulta particolarmente utile in contesti aziendali dove procedure standard devono essere ripetute con variazioni parametriche, permettendo di astrarre la complessità sottostante attraverso interfacce semplificate [Mod25c].

1.2.2 Primitive Bidirezionali e Interazione Utente

Oltre alle primitive esposte dai server, la specifica MCP definisce meccanismi che i server possono utilizzare per invocare funzionalità sul lato client, abilitando interazioni bidirezionali più sofisticate e mantenendo l'utente umano al centro del processo decisionale [Mod25b].

Il meccanismo di **Sampling** permette ai server di richiedere al client l'esecuzione di inferenze da parte dell'LLM senza essere direttamente accoppiati a un modello specifico. Questa astrazione è fondamentale in scenari dove il server necessita di capacità di elaborazione del linguaggio naturale, ad esempio per generare sintesi di documenti o estrarre informazioni strutturate da testo libero. Come illustrato in Figura 1.1, il flusso prevede che il server invii una richiesta di completamento al client, il quale la sottopone al modello linguistico. Prima di restituire il risultato al server, il sistema può richiedere l'approvazione esplicita dell'utente, implementando così un pattern di *human-in-the-loop* che garantisce supervisione e controllo sulle operazioni potenzialmente sensibili [Mod25b].

1.2. DEFINIZIONE E RUOLO DEL MODEL CONTEXT PROTOCOL (MCP)

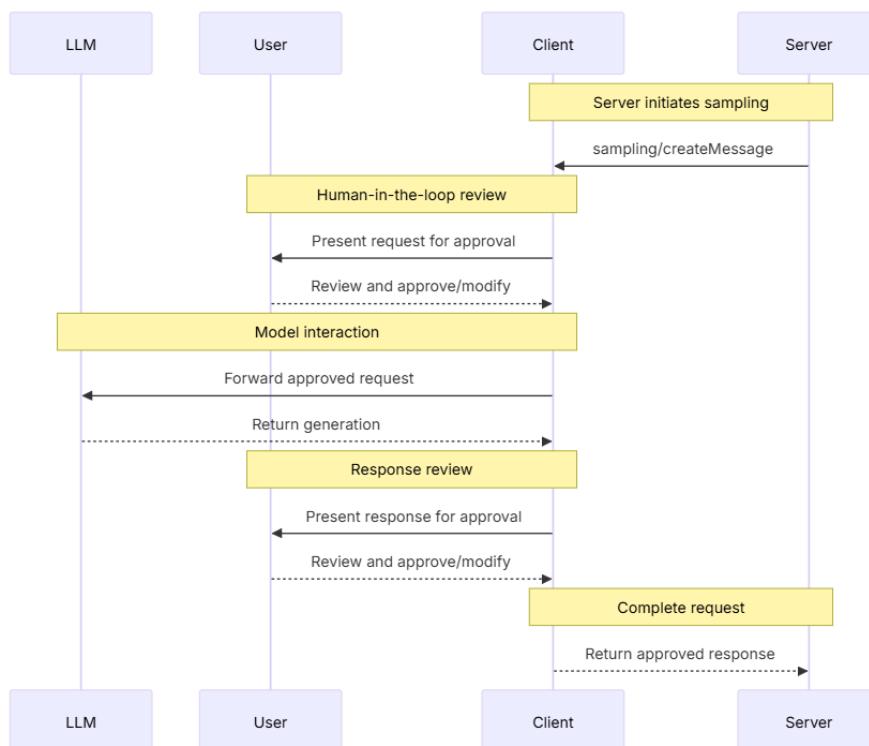


Figura 1.1: Architettura del primitive Sampling nel *MCP* che mostra il ciclo di richiesta–inferenza–approvazione tra client, server e utente umano.[Mod25b]

1.2. DEFINIZIONE E RUOLO DEL MODEL CONTEXT PROTOCOL (MCP)

La primitiva di **Elicitation** affronta invece la necessità di raccogliere input specifici dall'utente durante l'esecuzione di operazioni complesse. Quando un server decide che per completare un'azione è necessaria una conferma esplicita o un parametro aggiuntivo, può invocare il client richiedendo l'interazione diretta con l'utilizzatore. Questo pattern risulta essenziale in scenari transazionali come la finalizzazione di una prenotazione o l'autorizzazione di un pagamento, dove la conferma umana rappresenta un requisito necessario. La Figura 1.2 illustra il ciclo completo di richiesta, interazione con l'utente e restituzione della risposta al server [Mod25b].

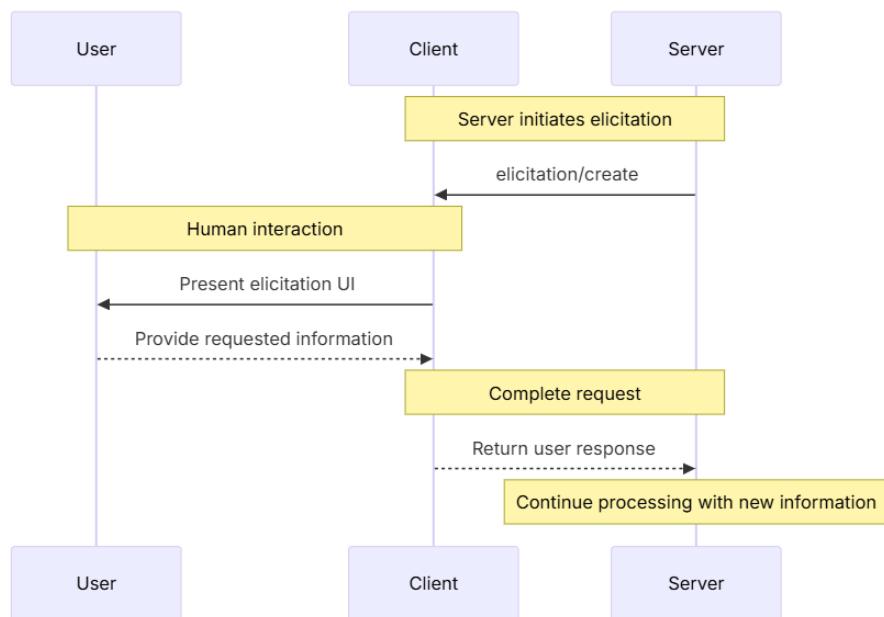


Figura 1.2: Architettura del primitive Elicitation nell'MCP che mostra il ciclo di richiesta-interazione umana-risposta tra Server, Client e utente.[Mod25b]

Il concetto di **Roots** introduce un meccanismo per definire i confini logici entro cui i server devono operare. Tipicamente utilizzato per determinare quali porzioni di filesystem o scope operativi specifici possono essere utilizzati, queste primitive guidano i server nella selezione delle risorse su cui concentrare le proprie attività, evitando accessi non autorizzati o sprechi computazionali dovuti all'analisi di dati irrilevanti [Mod25b].

1.2. DEFINIZIONE E RUOLO DEL MODEL CONTEXT PROTOCOL (MCP)

1.2.3 Vantaggi Sistemici dell'Adozione MCP

L'introduzione di uno standard condiviso per l'integrazione tra sistemi AI e risorse esterne genera benefici che trascendono il singolo progetto, impattando l'intero ecosistema di sviluppo e deployment di applicazioni intelligenti.

La **standardizzazione** costituisce il vantaggio più immediato e tangibile. Eliminando la necessità di sviluppare integrazioni proprietarie per ciascuna combinazione LLM-sistema esterno, il protocollo riduce drasticamente il numero di sforzi ingegneristici e permette agli sviluppatori di concentrarsi sulla logica di business piuttosto che sulla gestione dell'infrastruttura di comunicazione. Il codice di integrazione diventa riutilizzabile attraverso diverse applicazioni AI, accelerando il time-to-market e riducendo i costi di sviluppo [Mod25e, KD25].

Sull'aspetto della **sicurezza**, l'MCP promuove l'adozione di pattern consolidati per l'autenticazione, l'autorizzazione e l'audit delle operazioni. La presenza di meccanismi standardizzati riduce le vulnerabilità introdotte da implementazioni ad-hoc e facilita la conformità a requisiti normativi in settori regolamentati. La trasparenza delle interazioni, garantita dal logging strutturato delle chiamate a strumenti e risorse, supporta inoltre attività di troubleshooting e analisi forense [SEKK25, KD25].

La **composabilità** del protocollo abilita architetture modulari dove i componenti possono essere sviluppati, testati e scalati indipendentemente. Particolarmente rilevante è la capacità dei nodi di funzionare contemporaneamente come client e server, permettendo la costruzione di catene di agenti gerarchiche dove server intermedi aggregano e trasformano dati provenienti da server di livello inferiore prima di esporli ai client di livello superiore. Questa flessibilità architetturale supporta scenari complessi tipici di sistemi enterprise distribuiti [SEKK25, KD25].

Infine, dal punto di vista strategico, l'MCP si posiziona come fondamento di una roadmap evolutiva più ampia nell'ecosistema dei protocolli AI. La sua adozione rappresenta un primo passo verso standard più sofisticati per la coordinazione multi-agente (Agent-to-Agent protocols) e la gestione di workflow complessi, fungendo da base tecnica e concettuale per future iterazioni del panorama tecnologico [ESGK25].

1.3 WebSocket: Protocollo per Comunicazione Real-Time

Il protocollo WebSocket, standardizzato dall'IETF con la RFC 6455 nel dicembre 2011 [FM11], rappresenta la tecnologia di riferimento per la gestione di comunicazioni bidirezionali (*full-duplex*) e persistenti tra client e server. Nato con l'obiettivo di superare i limiti dell'architettura RESTful tradizionale, questo standard definisce un meccanismo di trasporto su TCP che, pur mantenendo la compatibilità con l'infrastruttura web esistente (come proxy e firewall), si discosta radicalmente dal modello richiesta-risposta tipico di HTTP [FM11].

Il ciclo di vita di una connessione WebSocket inizia con una fase di negoziazione, nota come *opening handshake*, che sfrutta lo stesso protocollo HTTP per stabilire il canale. Il client invia una richiesta GET contenente l'header `Upgrade: websocket`, segnalando l'intenzione di cambiare protocollo; se il server accetta la richiesta, risponde con lo stato `101 Switching Protocols` [FM11]. Terminato questo scambio iniziale, il protocollo HTTP viene abbandonato e la connessione TCP sottostante rimane aperta, andando a creare un tunnel persistente per lo scambio continuo di dati.

Tale approccio risolve le inefficienze storiche legate alle tecniche di *polling* o *long-polling*, dove il client era costretto a interrogare periodicamente il server o a mantenere aperte connessioni HTTP multiple in attesa di dati. WebSocket ha eliminando la necessità di instaurare nuove connessioni TCP e di inviare ridondanti header HTTP per ogni singolo messaggio, riducendo drasticamente l'overhead di rete e la latenza, aprendo a nuovi scenari real-time ad alte prestazioni come dashboard finanziarie, giochi multiplayer o sistemi di messaggistica istantanea [Mic23].

1.3.1 Caratteristiche architetturali

La peculiarità fondamentale del protocollo risiede nella sua natura *full-duplex*, che disaccoppia la comunicazione dalla logica di richiesta esplicita: client e server possono trasmettere dati in qualsiasi momento e in modo indipendente l'uno dall'altro [FM11] determinando una maggiore libertà operativa. Essa è supportata da una

1.3. WEBSOCKET: PROTOCOLLO PER COMUNICAZIONE REAL-TIME

struttura orientata ai messaggi (*message-oriented*), che si contrappone al flusso di byte continuo ("stream") tipico del TCP puro. I dati applicativi vengono incapsulati in unità discrete denominate *frame*, le quali possono trasportare payload testuali (codificati in UTF-8) o binari, semplificando notevolmente l'implementazione lato applicativo rispetto alla gestione diretta dei socket grezzi [FM11].

L'efficienza del protocollo è garantita da un framing estremamente leggero. A differenza dei pacchetti HTTP, che trasportano corposi metadati a ogni scambio, i frame WebSocket aggiungono solo pochi byte di intestazione al payload utile. Questo design minimizza il consumo di banda e riduce significativamente la latenza percepita rispetto alle soluzioni basate su polling, eliminando il ritardo introdotto dall'instaurazione ripetuta delle connessioni [Mic23]. Inoltre, il protocollo prevede frame di controllo specifici, come *Ping* e *Pong*, essenziali per monitorare lo stato di salute della connessione (heartbeat). Tramite tali meccanismi gli endpoint sono in grado di rilevare disconnessioni anomale e di mantenere attivo il canale anche in presenza di NAT o proxy che potrebbero chiudere le connessioni inattive, garantendo così la stabilità e l'affidabilità necessarie per le moderne applicazioni web [FM11].

1.3.2 Vantaggi operativi e prestazionali

L'adozione dello standard WebSocket comporta un significativo avanzamento rispetto alle architetture tradizionali basate su polling, offrendo benefici concreti in ambito di efficienza di rete che di reattività applicativa. Il vantaggio più immediato risiede nel drastico abbattimento della latenza *end-to-end*. Eliminando la necessità di negoziare ripetutamente la connessione e di trasmettere ridondanti header HTTP per ogni scambio di dati, il protocollo minimizza l'overhead strutturale successivo all'*handshake* iniziale. Le documentazioni tecniche confermano come questa ottimizzazione del payload garantisca un utilizzo della larghezza di banda strettamente limitato ai dati applicativi reali, superando le limitazioni prestazionali del *long-polling* [Mic23].

La peculiarità della leggerezza strutturale apporta dei vantaggi anche sulla scalabilità lato server. A differenza del polling, che invia al backend migliaia di richieste HTTP distinte, WebSocket mantiene attive singole connessioni TCP

1.3. WEBSOCKET: PROTOCOLLO PER COMUNICAZIONE REAL-TIME

persistenti. Questo modello riduce il carico computazionale necessario per l'apertura e chiusura dei socket, permettendo all'infrastruttura di gestire un numero elevato di connessioni concorrenti con un consumo di risorse contenuto [Mic23]. Tale caratteristica rende l'architettura particolarmente idonea per scenari ad alta densità di utenti, facilitando le strategie di scaling orizzontale.

Dal punto di vista funzionale, il protocollo abilita un vero e proprio modello di comunicazione *push* proattivo. Il server diviene in grado di trasmettere informazioni al client istantaneamente, non appena queste divengono disponibili, svincolandosi dalle logiche di richiesta esplicita. Tale cambio di paradigma è fondamentale per l'implementazione di funzionalità *real-time* moderne, come chat, notifiche istantanee o dashboard live, senza ricorrere a complessi meccanismi di sincronizzazione. Infine, l'affermazione dell'ecosistema garantisce un'ampia interoperabilità: il supporto nativo dell'API `WebSocket` in tutti i moderni browser e l'integrazione nei principali framework lato server (incluso l'ambiente .NET tramite SignalR [Mic25a]) rendono l'adozione di questa tecnologia uno standard de facto, privo della necessità di plugin esterni o soluzioni proprietarie.

1.3.3 Limitazioni intrinseche e sfide implementative

Sebbene il protocollo `WebSocket` offra innegabili vantaggi in termini di latenza e ridotto overhead, la sua adozione introduce complessità architettonali non trascurabili rispetto al modello HTTP stateless tradizionale. Una prima criticità riguarda la gestione delle risorse lato server, infatti la natura persistente della connessione impone che il server mantenga attivo lo stato per ogni client collegato, gestendo allocazione di memoria, socket aperti e meccanismi di *heartbeat* (ping/pong). In scenari ad alta concorrenza, ciò comporta un consumo di risorse significativamente superiore rispetto a un approccio stateless, richiedendo strategie di scalabilità orizzontale più sofisticate.

Un ulteriore ostacolo risiede nell'affidabilità della connessione in ambienti di rete eterogenei. Il protocollo, nella sua forma "grezza" (*raw*), non prevede meccanismi nativi di fallback, indispensabili qualora la connessione non possa essere stabilita a causa di proxy restrittivi, firewall aziendali o client obsoleti, la comunicazione fallisce senza alternative automatiche. Spetta dunque allo sviluppatore il compito

1.3. WEBSOCKET: PROTOCOLLO PER COMUNICAZIONE REAL-TIME

di implementare logiche personalizzate di riconnessione o di degradazione verso protocolli alternativi.

Infine, vanno considerati gli aspetti di sicurezza. L'utilizzo del protocollo su canali cifrati (`wss://`) è doveroso per proteggere l'integrità dei dati, introducendo tuttavia un costo computazionale aggiuntivo per le operazioni crittografiche TLS. Parallelamente, il modello di sicurezza differisce da quello delle richieste HTTP standard dove le policy CORS (Cross-Origin Resource Sharing) non si applicano direttamente all'handshake WebSocket, costringendo il server a validare esplicitamente l'header `Origin` per prevenire attacchi di tipo Cross-Site WebSocket Hijacking (CSWSH).

1.3.4 Adozione di SignalR nel contesto progettuale

Alla luce delle sfide operative sopra discusse, l'architettura del progetto realizzato in questa tesi ha previsto l'integrazione di SignalR come livello di astrazione per la gestione della comunicazione real-time. Questa scelta strategica permette di beneficiare delle prestazioni del protocollo WebSocket, utilizzato come trasporto preferenziale in conformità alla RFC 6455, alleggerendone al contempo le rigidità strutturali [Mic25a].

Il vantaggio determinante di SignalR risiede nella sua capacità di negoziare automaticamente il miglior trasporto disponibile. Qualora l'ambiente di rete non supporti WebSocket, la libreria esegue un fallback trasparente verso tecnologie alternative, come i Server-Sent Events (SSE) o il long-polling, garantendo la continuità del servizio senza richiedere interventi manuali nel codice applicativo. Delegando al framework la complessità della gestione delle connessioni, delle riconnessioni automatiche e della compatibilità trasversale, è stato possibile concentrare lo sviluppo sulla logica di business, aumentando la robustezza complessiva del sistema. I dettagli implementativi e la configurazione specifica di SignalR all'interno dello stack tecnologico verranno analizzati approfonditamente nel Capitolo 3.

1.4 Panoramica e contesto su i Configuratori 3D

Nel contesto dell'informatica applicata ai processi industriali e commerciali, la configurazione di prodotto rappresenta un'attività progettuale specifica che interviene a valle della definizione ingegneristica primaria [Zha14]. Essa può essere definita come il processo di composizione di un prodotto personalizzato partendo da un insieme predefinito di componenti, parti o assemblaggi, nel rispetto di un set di vincoli ben determinati che limitano le modalità di selezione e combinazione degli elementi stessi [Zha14]. L'obiettivo finale di tale processo, supportato da sistemi software denominati configuratori, è la generazione di una specifica di prodotto valida, che includa distinta base (BOM), parametri di design e relazioni strutturali [Zha14].

L'evoluzione verso la *Mass Customization* ha reso necessario lo sviluppo di piattaforme che permettano al cliente di interagire direttamente con lo "spazio delle soluzioni" del prodotto [HLE09]. In questo scenario, i configuratori 3D assumono un ruolo cruciale, non limitandosi alla sola selezione delle opzioni proposte, ma offrendo una visualizzazione in tempo reale delle scelte effettuate. La letteratura identifica nella rappresentazione visiva un elemento critico, infatti, mentre i configuratori tradizionali possono limitarsi a descrizioni testuali o bidimensionali, le moderne piattaforme web-based integrano tecnologie di visualizzazione 3D per colmare il divario tra le aspettative del cliente e il prodotto finale [AR12]. Tuttavia, l'efficacia di tali sistemi dipende strettamente dalla capacità di gestire la complessità computazionale e la trasmissione dei dati su reti web, spesso ricorrendo a tecnologie come WebGL o viewer basati su plugin (ad esempio Google Earth API o standard CityGML) per il rendering della scena [AR12].

1.4.1 Gestione delle Regole e Motore di Configurazione

Il cuore funzionale di un configuratore non risiede solamente nella sua interfaccia grafica, ma nel motore logico che regola la coerenza del prodotto. La configurazione opera all'interno di uno spazio di soluzioni predefinito, caratterizzato da processi stabili ma flessibili [HLE09]. Per gestire questa complessità, il sistema deve contenere conoscenze del dominio organizzate in ontologie, vincoli e regole [Zha14].

Le regole di configurazione permettono di porre un controllo sulla fattibilità tecnica definendo le relazioni tra i componenti (ad esempio, "Se viene selezionato il Componente A, allora il Componente B è obbligatorio") e prevengono combinazioni invalide. In contesti ingegneristici complessi (*Engineer-To-Order*), la sfida principale nella progettazione del configuratore risiede proprio nella costruzione di una base di conoscenza (*Knowledge Base*) robusta, capace di automatizzare compiti di specificazione che tradizionalmente richiederebbero l'intervento umano [HLE09]. Modellare tale conoscenza può avvalersi di approcci basati su *Constraint Satisfaction Problems* (CSP), che permettono al sistema di verificare la consistenza delle scelte dell'utente e, se necessario, proporre spiegazioni o diagnosi in caso di conflitti [Zha14].

1.4.2 Calcolo Dinamico e Output del Processo

Parallelamente alla validazione tecnica, il configuratore deve gestire gli aspetti economici e documentali della transazione. Sebbene in un contesto di *Mass Customization* l'obiettivo sia mantenere i prezzi vicini a quelli della produzione di massa, nei sistemi complessi il configuratore agisce spesso come strumento di automazione per la generazione di preventivi (quotazioni) [HLE09].

Il calcolo dinamico del prezzo avviene in tempo reale in risposta alle modifiche apportate dall'utente alla configurazione 3D. Il sistema associa a ogni componente o variante selezionata un valore economico, aggiornando istantaneamente il totale. Al termine del processo, l'output non è limitato alla sola visualizzazione a schermo ma il configuratore deve essere in grado di esportare dati strutturati necessari per la produzione e la vendita. Essi includono tipicamente la distinta base (BOM), i piani di instradamento (*routing plans*), le specifiche tecniche dettagliate e la documentazione di offerta [Zha14]. In alcuni scenari avanzati, come nel settore dell'impiantistica o dell'ingegneria civile, l'automazione riguarda la creazione di specifiche di prodotto complete che fungono esse stesse da output primario del processo [HLE09].

1.4.3 Architettura del Sistema: Frontend, Backend e Scambio Dati

Dal punto di vista dell'implementazione tecnica, un configuratore 3D moderno adotta tipicamente un'Architettura Orientata ai Servizi (SOA), strutturata in livelli concettuali distinti che garantiscono modularità e scalabilità [AR12].

- Il livello di **Frontend**: (o *Presentation Layer*) è responsabile dell'interfaccia utente e della visualizzazione 3D. Implementato tramite tecnologie web standard (XHTML dinamico, Ajax, JavaScript), questo livello gestisce l'integrazione con l'utente e comunica con il server per aggiornare la scena [AR12]. La visualizzazione 3D richiede la gestione efficiente di oggetti geometrici complessi per questo è essenziale ottimizzare il formato di scambio dati tra client e server. Sebbene l'XML (*eXtensible Markup Language*) sia stato storicamente utilizzato, è stato osservato che esso comporta un *overhead* eccessivo per la trasmissione di grandi moli di dati geometrici. Di conseguenza, formati più compatti come JSON (*JavaScript Object Notation*) sono preferibili per garantire prestazioni elevate e ridurre i tempi di caricamento nel viewer 3D [AR12].
- Il **Backend** ospita il motore della piattaforma (*Platform Engine* o *Business Layer*). Questo livello contiene i componenti core che eseguono la logica di business, gestiscono il *workflow* di configurazione e coordinano l'analisi dei modelli [AR12]. È in questo strato che risiede il *Model Coordinator*, responsabile di orchestrare gli input e gli output dei vari moduli di calcolo e assicura che i dati scambiati tra le componenti siano semanticamente coerenti.
- Infine, il livello **Database** (o *Data Layer*) costituisce il repository per la persistenza dei dati. In applicazioni che richiedono la gestione di oggetti spaziali e modelli 3D, è necessario utilizzare *Database Management Systems* (DBMS) capaci di gestire dati spaziali e non spaziali, timestamp e metadati relativi alle librerie di modelli [AR12]. L'accesso a questi dati è gestito da un livello di accesso ai dati (*Data Access Layer*) in grado di diminuire la complessità delle query dirette al database grazie all'esposizione di API per il recupero e la manipolazione degli oggetti necessari alla configurazione. Questa

separazione garantisce che la logica di visualizzazione nel frontend rimanga disaccoppiata dalla struttura fisica di memorizzazione dei dati, facilitando la manutenzione e l’evoluzione futura del sistema [AR12].

1.4.4 Casi d’uso e applicazioni dei configuratori 3D

L’adozione di configuratori 3D nell’ambito della mass customization rappresenta un punto di svolta nell’interazione tra aziende e consumatori, andando a creare un valore sia per chi acquista sia per chi produce. L’analisi della letteratura scientifica permette di comprendere come questi strumenti si inseriscano in diversi contesti applicativi, con impatti concreti su molteplici dimensioni del processo di acquisto e produzione.

La mass customization, definita come la capacità di fornire prodotti personalizzati a ogni cliente attraverso agilità, flessibilità e integrazione di processo [FSB04], trova nei configuratori 3D uno degli strumenti più efficaci per conciliare la produzione di massa con le esigenze individuali dei clienti. L’obiettivo primario è fornire prodotti che rispondano al meglio alle scelte reali dei clienti, basate su necessità e preferenze specifiche, mantenendo al contempo un’efficienza prossima a quella della produzione di massa [FSB04]. I configuratori 3D si collocano in questo contesto come mediatori tra la complessità tecnica del design modulare e la semplicità richiesta dall’esperienza utente.

Uno dei settori in cui i configuratori 3D hanno dimostrato particolare efficacia è quello della personalizzazione di prodotti complessi e costosi, come nel caso della configurazione di appartamenti residenziali. In tale ambito, i configuratori permettono ai clienti di navigare tra componenti e opzioni offerte dall’azienda, definendo vincoli e regole relative alle configurazioni possibili del prodotto [FSB04]. La modularità dei prodotti, basata sulla composizione di componenti standard o sulla variazione controllata di elementi predefiniti, favorisce l’uso di interfacce 3D che rendono le scelte immediatamente comprensibili.

Dal **punto di vista del cliente**:

I configuratori 3D rispondono a una lacuna fondamentale dei siti di e-commerce tradizionali, ovvero la carenza di esperienze di acquisto che si avvicinino alla realtà. Le interfacce 2D, basate su immagini statiche e testo, non permettono ai

1.4. PANORAMICA E CONTESTO SU I CONFIGURATORI 3D

consumatori di sviluppare una sensazione di presenza, interattività e coinvolgimento collaborativo [EAAS17]. Al contrario, gli ambienti 3D simulano spazi reali, offrono modelli di prodotto tridimensionali e consentono ai clienti di navigare virtualmente attraverso le proprie scelte, replicando dinamiche tipiche dei negozi fisici. Questa esperienza immersiva migliora significativamente la qualità percepita dell’interazione e la soddisfazione complessiva del consumatore.

In particolare, l’integrazione di tecnologie di realtà virtuale (VR) negli ambienti di e-commerce 3D consente ai clienti di sperimentare i prodotti in modo più realistico prima dell’acquisto, superando i limiti delle recensioni scritte o delle valutazioni numeriche. La possibilità di testare virtualmente prodotti prima della fase di acquisto riduce l’incertezza e favorisce una valutazione più accurata delle alternative disponibili [EAAS17]. Inoltre, l’uso di strumenti interattivi basati su avatar e agenti virtuali contribuisce a creare un senso di presenza sociale e fiducia, elementi cruciali nei contesti di commercio online [EAAS17].

L’efficacia dei configuratori 3D si estende anche alla gestione della complessità decisionale. Quando i clienti sono chiamati a configurare un prodotto selezionando tra un numero elevato di componenti e opzioni, il processo decisionale può diventare complesso e frustrante, specialmente per utenti non esperti. I configuratori avanzati integrano modelli decisionali multicriterio che supportano il cliente nell’identificare la configurazione ottimale in base alle proprie preferenze e vincoli. Ad esempio, un sistema decisionale che utilizza tecniche di programmazione lineare intera può massimizzare l’utilità del cliente soggetta a vincoli tecnici, estetici e finanziari definiti sia dall’utente sia dai progettisti [FSB04]. In questo modo, il configuratore non si limita a visualizzare le opzioni, ma guida attivamente l’utente verso scelte che rispondono al meglio alle sue esigenze.

Dal lato aziendale:

L’adozione di configuratori 3D offre vantaggi strategici e operativi significativi. In primo luogo, questi strumenti facilitano l’acquisizione di conoscenza tacita dai clienti, trasformandola in informazioni esplicite e strutturate che possono essere integrate nei database aziendali. Il processo di configurazione genera dati preziosi sulle preferenze individuali, sulle funzioni di valore assegnate agli attributi dei prodotti e sui vincoli espressi dai consumatori. Questa base di conoscenza costituisce un patrimonio informativo che l’azienda può utilizzare per orientare lo sviluppo di

1.4. PANORAMICA E CONTESTO SU I CONFIGURATORI 3D

nuovi prodotti e per affinare le strategie di marketing [FSB04].

Inoltre, i configuratori 3D contribuiscono a ridurre i costi di personalizzazione, rendendo la mass customization economicamente sostenibile. La modularità, supportata da rappresentazioni compatte e gerarchiche delle architetture di prodotto consente alle aziende di offrire varianti in modo economicamente sostenibile. L'adozione di modelli object-oriented e algoritmi di ottimizzazione permette di affrontare la complessità derivante dalle combinazioni, mantenendo coerenza tecnica e fattibilità produttiva [FSB04].

Un altro beneficio rilevante per le aziende riguarda il miglioramento della fiducia dei consumatori. Nei contesti di e-commerce tradizionale, la costruzione della fiducia è ostacolata dalla mancanza di interazione fisica e dalla difficoltà di valutare la qualità dei prodotti. I configuratori 3D, attraverso la simulazione realistica e la visualizzazione dettagliata, aumentano la trasparenza del processo di acquisto e facilitano la formazione di fiducia cognitiva ed emotiva nei confronti del fornitore [EAAS17]. La presenza di avatar che fungono da consulenti virtuali e l'uso di interfacce vocali contribuiscono ulteriormente a migliorare la percezione di affidabilità e competenza dell'azienda.

Nel settore del marketing, i configuratori 3D si configurano come strumenti promozionali altamente efficaci. La possibilità di coinvolgere i clienti in esperienze immersive e interattive aumenta il livello di engagement e rafforza l'equità del brand. La ricchezza dell'ambiente virtuale e il grado di coinvolgimento che esso genera hanno un impatto positivo sull'intenzione di acquisto e sulla fedeltà del cliente [EAAS17]. Le aziende possono sfruttare questi strumenti per condurre test di mercato su prototipi virtuali di nuovi prodotti, raccogliendo feedback dai consumatori in modo rapido e meno oneroso rispetto ai test su prodotti fisici.

L'applicazione dei configuratori 3D si estende anche a settori che richiedono elevata personalizzazione tecnica, come la produzione di biciclette su misura, l'arredamento personalizzato e la progettazione di componenti biomeccanici. In questi ambiti, la combinazione di modelli 3D interattivi con sistemi di supporto decisionale consente ai clienti di esprimere le proprie esigenze in termini di attributi funzionali ed estetici, lasciando al sistema il compito di tradurre tali esigenze in specifiche tecniche realizzabili. Questo approccio riduce il rischio di errori di configurazione e accelera il processo di definizione del prodotto finale.

Tuttavia, l'implementazione di configuratori 3D presenta delle tortuosità da gestire. La complessità del processo di valutazione aumenta significativamente al crescere del numero di componenti, opzioni e criteri di valutazione. Anche con l'ausilio di strumenti grafici e interattivi, l'analisi delle funzioni di valore, la definizione delle condizioni di indipendenza tra criteri e il confronto tra numerose alternative possono risultare compiti onerosi per il cliente [FSB04]. È quindi essenziale progettare interfacce che bilancino espressività e semplicità d'uso, guidando l'utente senza sovraccaricarlo di informazioni.

I configuratori 3D rappresentano una nuova tecnologia avanzata in grado di rispondere alle esigenze di personalizzazione del mercato contemporaneo. Per i clienti, essi offrono esperienze di acquisto più ricche, trasparenti e soddisfacenti, riducendo l'incertezza e facilitando scelte consapevoli. Per le aziende, costituiscono strumenti strategici per la raccolta di conoscenza, l'ottimizzazione dei processi produttivi, la riduzione dei costi di personalizzazione e il rafforzamento del legame con il cliente. La loro diffusione nei prossimi anni dipenderà dalla capacità di integrare efficacemente tecnologie di realtà virtuale, modelli decisionali avanzati e architetture di prodotto modulari, mantenendo al contempo un'interfaccia accessibile e intuitiva per utenti con diversi livelli di competenza tecnica.

1.4.5 Sfide dell'Integrazione AI-3D

L'integrazione tra sistemi di Intelligenza Artificiale Conversazionale e configuratori 3D costituisce un dominio ingegneristico complesso, caratterizzato dalla necessità di far convergere l'elasticità dell'elaborazione del linguaggio naturale (NLP) con la rigidità dei vincoli spaziali e parametrici tipici della progettazione assistita. In questo scenario, il Model Context Protocol (MCP) assume il ruolo cruciale di meccanismo di orchestrazione e intermediazione, necessario per colmare il divario architettonurale tra un Large Language Model (LLM) e un ambiente di simulazione altamente strutturato.

Una delle criticità primarie risiede nella traduzione affidabile dell'intento utente in comandi macchina logicamente validi. Nonostante gli LLM siano strumenti potenti per convertire il linguaggio naturale in rappresentazioni simboliche o chiamate API [ROH25, SDYD⁺23, PZWG24], la garanzia della correttezza sintattica rimane

una sfida aperta [ROH25]. Per mitigare il rischio di output non conformi, la letteratura propone tecniche come il Grammar Constrained Decoding (GCD), che obbliga il modello a generare output aderenti a una grammatica formale predefinita, come una Context-Free Grammar [ROH25]. Tuttavia, sebbene il GCD possa assicurare la validità formale della sintassi, non è in grado di prevenire errori semanticici, come violazioni della consistenza dei predicati o riferimenti a variabili inesistenti, che impediscono l'esecuzione corretta delle istruzioni nel configuratore [ROH25].

Oltre alla traduzione dei comandi, emerge la complessità legata alla sincronizzazione dello stato e alla gestione dei vincoli in un ambiente distribuito. Il configuratore 3D opera come uno strumento specializzato che richiede input precisi per eseguire calcoli complessi [PZWG24]. Di conseguenza, l'MCP deve garantire che le richieste generate dall'AI rispettino rigorosamente i limiti interni del motore 3D, come la precisione numerica o i range validi dei parametri [PZWG24]. Questo richiede un netto disaccoppiamento architettonico: le attività di parsing e ragionamento sono delegate all'LLM, mentre l'esecuzione effettiva spetta al configuratore, con l'MCP che funge da orchestratore centrale del flusso di lavoro [DZY22, ROH25, SDYD⁺23]. Tale separazione preserva la generalità decisionale del modello linguistico, che deve apprendere autonomamente quali strumenti invocare e come popolarne gli argomenti [SDYD⁺23].

Tuttavia, la gestione di transazioni che coinvolgono servizi eterogenei introduce sfide significative per la coerenza dei dati. In caso di errori durante l'elaborazione, ad esempio un vincolo geometrico non soddisfatto, il sistema deve essere in grado di gestire il fallimento e avviare azioni di compensazione per annullare eventuali modifiche parziali [DZY22]. Per evitare anomalie dovute alla mancanza di isolamento dei dati, è preferibile posporre il commit finale sul database fino al completamento dell'intero flusso, gestendo lo stato transitorio a livello di cache in memoria [DZY22]. L'uso di sistemi di messaging asincrono, come Apache Kafka, risulta essenziale in questo contesto per mantenere l'ordine delle richieste e garantire la consistenza finale [DZY22].

Questo livello intermedio comporta un aumento di latenza che può degradare l'esperienza realtime [SDYD⁺23]. Per gestire scenari ad alto traffico sono necessarie strategie come l'impiego di cache in memoria (per esempio Redis), che riducono l'I/O su disco e migliorano il throughput delle operazioni CRUD [DZY22]. Inoltre,

1.4. PANORAMICA E CONTESTO SU I CONFIGURATORI 3D

la multimodalità richiede che segnali testuali e visivi vengano allineati e fusi correttamente per risolvere riferimenti spaziali nella scena 3D [SH22].

Infine, l'integrazione di componenti distribuiti richiede una rigorosa gestione della sicurezza, in particolare per l'autenticazione cross-system. L'utilizzo di token JWT (JSON Web Token) è lo standard per l'accesso alle risorse, ma la natura stateless di questi strumenti rende complessa la loro revoca immediata in caso di compromissione [RR25]. È dunque compito dell'MCP interagire con sistemi di monitoraggio applicativo per rilevare anomalie nei comportamenti, come il token hijacking o tentativi di generazione illecita di credenziali, garantendo l'integrità del canale di comunicazione tra l'intelligenza artificiale e il motore di configurazione [RR25].

Capitolo 2

Architettura Tecnologica del Progetto

Questo capitolo descrive le tecnologie utilizzate nel progetto, spiegando per ognuna cosa fa e perché è stata scelta. L'obiettivo è fornire le conoscenze necessarie per comprendere l'architettura senza perdersi in dettagli implementativi, che saranno affrontati nei capitoli successivi dedicati alla progettazione e all'implementazione.

La struttura segue la logica dell'architettura del sistema. Si inizia con la piattaforma di backend: .NET 8 come runtime, C# 12 come linguaggio di programmazione, ASP.NET Core per l'infrastruttura web e l'SDK MCP per comunicare con i modelli di intelligenza artificiale. Successivamente si analizzano i protocolli di comunicazione: REST per le operazioni standard, JSON-RPC per le chiamate di procedura remota e WebSocket per le connessioni persistenti. SignalR viene presentato come soluzione per gestire la comunicazione in tempo reale, nascondendo la complessità dei protocolli sottostanti e offrendo meccanismi automatici di riconnessione.

La sezione sulla sicurezza esamina come il sistema protegge le comunicazioni: i token JWT permettono l'autenticazione senza mantenere stato sul server, CORS regola l'accesso tra domini diversi e TLS critta i dati in transito. Il capitolo si chiude con l'integrazione frontend, dove TypeScript garantisce la sicurezza dei tipi anche lato client e il client SignalR gestisce la comunicazione con il server.

Ogni tecnologia viene presentata con un approccio diretto: prima si spiega

cos'è, poi perché è stata preferita alle alternative, infine quale ruolo svolge nel progetto. Le tabelle comparative facilitano il confronto tra opzioni diverse, mentre i diagrammi chiariscono i flussi di comunicazione.

2.1 Panoramica dello Stack

La definizione dell'architettura software per un sistema distribuito che integri capacità di intelligenza artificiale conversazionale con ambienti di configurazione tridimensionale richiede una selezione tecnologica rigorosa. La scelta dello stack non risponde solamente a criteri di popolarità o tendenza, bensì a una precisa strategia ingegneristica con l'obiettivo di garantire robustezza, manutenibilità e prestazioni elevate. L'ecosistema tecnologico adottato per il progetto è stato strutturato secondo una logica a livelli funzionali (*layered architecture*), progettata per disaccoppiare le responsabilità e facilitare l'evoluzione futura del sistema.

Il nucleo dell'infrastruttura di backend è costituito dalla piattaforma .NET 8 (LTS), supportata dal linguaggio C# 12 e dal framework ASP.NET Core. Questa combinazione rappresenta le fondamenta del sistema, offrendo un ambiente di esecuzione gestito ma altamente performante, capace di supportare sia le logiche di business complesse del protocollo *MCP* sia la gestione concorrente delle connessioni utente. Al di sopra del runtime, il livello di comunicazione agisce come sistema nervoso del progetto. Per soddisfare la necessità di interazioni bidirezionali a bassa latenza, è stato adottato SignalR, una libreria che astrae la complessità dei Web-Socket, gestendo automaticamente la negoziazione del trasporto. Parallelamente, l'integrazione con gli agenti cro:aiAI e il protocollo *MCP* sfrutta JSON-RPC 2.0, uno standard leggero per la chiamata di procedure remote, che convive con gli endpoint REST tradizionali utilizzati per operazioni di gestione meno frequenti.

La sicurezza attraversa trasversalmente questi livelli. L'autenticazione è viene gestita tramite lo standard JWT (JSON Web Token, RFC 7519), che permette una verifica dell'identità *stateless*, essenziale per la scalabilità orizzontale. La protezione del trasporto è garantita dalla crittografia TLS (HTTPS/WSS), mentre le politiche CORS regolano l'accesso alle risorse da origini differenti, scenario tipico nelle moderne applicazioni web distribuite. Infine, l'integrazione frontend è realizzata mediante TypeScript 5.x. L'utilizzo del pacchetto `@microsoft/signalr`

2.1. PANORAMICA DELLO STACK

permette al client web di instaurare un canale diretto con il server, beneficiando della tipizzazione statica che si estende dal backend al frontend, riducendo drasticamente le possibilità di errore durante lo scambio dei messaggi.

La convergenza verso questo specifico stack tecnologico è il risultato di una valutazione basata su diversi aspetti principali come: modernità, prestazioni, sicurezza dei tipi, maturità dell'ecosistema, interoperabilità e manutenibilità. L'adozione di una versione con supporto a lungo termine come .NET 8, rilasciata tra il 2023 e il 2024, permette di ricevere aggiornamenti di sicurezza per l'intero ciclo di vita del progetto che conferiscono grande stabilità. Sul fronte delle prestazioni, il runtime di .NET ha dimostrato negli ultimi anni miglioramenti significativi nel *throughput* delle richieste e nella gestione della memoria, requisiti imprescindibili per mantenere una latenza inferiore ai 10ms nelle operazioni di sincronizzazione dello stato del configuratore 3D.

Un fattore determinante è la garanzia di type-safety end-to-end dove l'impiego congiunto di C# lato server e TypeScript lato client permette di condividere definizioni di tipi Data Transfer Objects (DTO), spostando molte classi di errore dal runtime alla fase di compilazione. La maturità dell'ecosistema, librerie consolidate e documentazione esaustiva, riduce i rischi implementativi e accelera lo sviluppo tramite strumenti avanzati quali debugger, profiler, IntelliSense. L'adozione di standard aperti facilita interoperabilità e manutenibilità. La compatibilità nativa di .NET con JSON-RPC semplifica l'implementazione del Model Context Protocol, mentre pattern consolidati come la Dependency Injection in ASP.NET Core mantengono il codice modulare e testabile.

L'architettura delineata risponde puntualmente alle specifiche funzionali e non funzionali del sistema. Il requisito di interattività in tempo reale per il configuratore 3D trova in SignalR la soluzione ideale: la capacità di inviare notifiche *push* dal server al client permette di aggiornare la scena tridimensionale istantaneamente in risposta alle elaborazioni dell'cro:ai:AI, senza costringere il client a costosi meccanismi di polling. Per quanto concerne la scalabilità, l'approccio *stateless* garantito dai token JWT consente al sistema di crescere orizzontalmente senza la necessità di sincronizzare le sessioni utente tra nodi diversi. Inoltre, l'integrazione del *Model Context Protocol* è supportata nativamente dalle capacità di serializzazione JSON e dalla gestione dei trasporti, sia HTTP che stdio, offerte dal framework,

semplificando la creazione di strumenti personalizzati per i modelli linguistici.

La tabella 2.1 riassume le tecnologie selezionate per ogni livello logico, evidenziando le motivazioni specifiche che ne hanno determinato l'adozione nel contesto di questa tesi.

Tabella 2.1: Sintesi dello Stack Tecnologico e motivazioni di adozione.

Layer	Tecnologia	Versione	Motivazione Principale
Runtime	.NET	8 LTS	Performance elevate, stabilità LTS ed ecosistema maturo
Linguaggio	C#	12	Type-safety robusta e costrutti asincroni moderni
Framework Web	ASP.NET Core	8.0	Hosting SignalR integrato e Dependency Injection nativa
Integrazione MCP	MCP .NET SDK	0.6.0	Definizione tipizzata dei tool per l'AI
Real-Time Comm.	SignalR	8.0	Gestione automatica riconnessioni e fallback dei trasporti
Autenticazione	JWT Bearer	RFC 7519	Architettura stateless e scalabile per sistemi distribuiti
Sicurezza	CORS	W3C	Abilitazione sicura delle richieste cross-origin dal frontend
Sicurezza Trasp.	HTTPS/WSS	TLS 1.3	Cifratura end-to-end del canale di comunicazione
Linguaggio Front.	TypeScript	5.x	Coerenza dei tipi con il backend e supporto IntelliSense
SDK Frontend	@microsoft/signalr	8.x	Client leggero per la gestione eventi real-time nel browser

2.2 Piattaforma Backend .NET

2.2.1 .NET 8 e C# 12: Piattaforma e Linguaggio

L'adozione della piattaforma .NET 8 si colloca all'interno di un ecosistema software maturo, unificato e altamente ottimizzato [CR24a]. Dopo aver superato la storica separazione tra il framework proprietario orientato a Windows e le prime versioni modulari di .NET Core, Microsoft ha avviato con .NET 5 un percorso di convergenza architettonica, giunto a compimento nel novembre 2023 con il rilascio della versione 8, dotata di supporto a lungo termine (LTS) [Tou23, CR24a]. Questa evoluzione ha trasformato .NET in una soluzione interamente *open-source* e *cross-platform*,

progettata nativamente per operare con la medesima efficacia su ambienti Windows, Linux e macOS.

La forte efficienza di questo stack tecnologico risiede nei componenti avanzati come il CoreCLR e il *Garbage Collector* (GC). Il runtime implementa strategie di compilazione sofisticate. il compilatore Just-In-Time (JIT) sfrutta la *Dynamic Profile-Guided Optimization* (PGO) e la *Tiered Compilation* per adattare l'esecuzione del codice in base ai dati raccolti in tempo reale, minimizzando l'overhead e massimizzando il throughput [Tou23]. Parallelamente, il GC è stato oggetto di continue ottimizzazioni mirate alla riduzione dei tempi di pausa, fattore determinante per sostenere carichi di lavoro ad alta concorrenza tipici delle moderne applicazioni web basate su ASP.NET Core [Tou23].

Inoltre, il linguaggio C# 12 offre un'interfaccia espressiva, orientata agli oggetti e rigorosamente *type-safe* [Mic24e] stabilendo una stretta sinergia con il runtime. Le innovazioni introdotte in questa versione mirano a unire la sinteticità del codice con la sicurezza e le prestazioni. L'estensione dei Primary Constructors a classi e struct, unitamente alle Collection Expressions, permette di ridurre drasticamente il codice *boilerplate* necessario per la definizione di tipi e l'inizializzazione delle strutture dati [Mic24e]. Sul fronte dell'ottimizzazione, l'introduzione degli Inline Arrays e dei parametri `ref readonly` consente agli sviluppatori di gestire buffer a dimensione fissa e passaggi di riferimento in sola lettura con efficienza paragonabile a contesti *unsafe*, ma mantenendo le garanzie di sicurezza della memoria gestita [Tou23, Mic24e].

La scelta di combinare .NET 8 e C# 12 risponde a precisi requisiti di scalabilità e manutenibilità. Il supporto nativo al paradigma asincrono, tramite i costrutti `async/await`, e l'integrazione di LINQ (*Language Integrated Query*) offrono un modello ergonomico per la gestione dell'I/O non bloccante e delle interrogazioni in-memory, evitando complessità sintattiche e favorendo la pulizia del codice [Mic24e].

Sotto il profilo prestazionale, i benchmark evidenziano miglioramenti significativi nelle operazioni *compute-intensive*, come l'hashing crittografico, e una riduzione delle allocazioni di memoria nei percorsi critici [Tou23]. Tale efficienza, unita alla robustezza del sistema di tipi che previene intere classi di errori a tempo di compilazione [Mic24e], rende la piattaforma superiore rispetto ad alternative interpretate

per scenari real-time. Infine, la maturità dell'ecosistema, supportata dal gestore di pacchetti NuGet e dalla piena compatibilità con la containerizzazione Docker, garantisce un ciclo di sviluppo e rilascio affidabile, essenziale per l'integrazione di librerie complesse come gli SDK per protocoli distribuiti [CR24a].

2.2.2 ASP.NET Core: Infrastructure Framework

ASP.NET Core rappresenta l'evoluzione open-source, multipiattaforma e ad alte prestazioni del framework web di Microsoft, progettato per rispondere alle esigenze di scalabilità delle moderne architetture cloud-native [Mic25b, RJT24]. Distaccandosi radicalmente dal precedente ASP.NET, questa versione propone una riprogettazione strutturale con lo scopo di fornire una piattaforma unificata e leggera [KU18], in grado di sostenere lo sviluppo di sistemi distribuiti complessi e di gestire carichi di lavoro enterprise con efficienza elevata [Mic25b, JGVJ25].

L'architettura del framework è intrinsecamente modulare, caratteristica che consente di ridurre l'overhead operativo includendo nel processo solo i componenti strettamente necessari [RJT24]. Elemento centrale di questo design è la pipeline di gestione delle richieste HTTP, realizzata tramite middleware configurabili che intercettano e processano il traffico per implementare logiche trasversali come il logging e l'autenticazione [Mic25b, JGVJ25]. A livello di hosting, il sistema si avvale di Kestrel, un server web leggero e ottimizzato per l'esecuzione cross-platform, che supera la dipendenza storica da IIS garantendo prestazioni elevate anche in ambienti non Windows [Mic25b, KU18]. La robustezza dell'infrastruttura è ulteriormente assicurata dal supporto nativo per la dependency injection (DI), un pattern essenziale per favorire il disaccoppiamento dei componenti, la testabilità del codice e la manutenibilità a lungo termine [Mic25b, RJT24].

ASP.NET Core si distingue perché il suo runtime è più leggero e richiede meno memoria, caratteristiche che lo rendono molto veloce nell'elaborare le richieste secondo diversi benchmark del settore [RJT24, KU18]. Anche se componenti relativamente nuovi come il server Kestrel possono necessitare di tempo per essere pienamente consolidati, l'affidabilità dell'intero framework è dimostrata dall'uso su larga scala in servizi critici come Bing, Xbox e Azure [Mic25b, KU18]. Per questi

motivi, ASP.NET Core è una scelta appropriata quando si progettano applicazioni che devono garantire sia buone prestazioni sia stabilità operativa [Mic25b, RJT24].

Nel progetto ASP.NET Core è usato principalmente come infrastruttura interna, non come host principale delle API REST. Il framework viene impiegato per orchestrare i servizi interni tramite il contenitore per la dependency injection e per gestire funzionalità trasversali importanti come la configurazione basata sull'ambiente, il tracciamento (tracing) e il logging strutturato delle metriche di runtime [Mic25b, RJT24]. Inoltre, il server Kestrel è efficiente e viene usato per ospitare il componente SignalR di SignalrHub, permettendo comunicazioni real-time fluide e reattive, utili per l'interattività dell'applicazione [Mic25b].

2.2.3 MCP SDK: Astrazione Tool Definition

Il Model Context Protocol (MCP) si configura come uno standard aperto e ampiamente adottato, progettato per razionalizzare l'architettura dei sistemi basati su intelligenza artificiale generativa e facilitare l'integrazione tra i diversi componenti software. La sua funzione primaria è quella di definire uno schema rigoroso per strutturare le interazioni tra client e server, standardizzando le chiamate API verso i Large Language Models (LLM), le sorgenti dati e gli strumenti agentici, riducendo così significativamente i costi e la complessità dello sviluppo [RH25, ESGK25].

Per permettere l'uso del protocollo in diversi ambienti di sviluppo, sono stati rilasciati SDK open-source per i principali linguaggi di programmazione, tra cui Python, Java, TypeScript e Kotlin [RH25]. Nel contesto specifico di questo progetto, sviluppato in ambiente Microsoft, la componente tecnologica centrale è rappresentata dall'SDK ufficiale C# per MCP, mantenuto in stretta collaborazione con Microsoft [MM25]. Questo strumento, distribuito tramite il pacchetto NuGet `ModelContextProtocol`, funge da framework per l'implementazione di server MCP all'interno di applicazioni e librerie .NET, offrendo le estensioni necessarie per l'hosting e l'injection delle dipendenze [MM25]. La libreria è ideale per progetti che non hanno bisogno di un server HTTP completo. Si basa sulla classe `McpServer`, configurabile sia in modo esplicito che usando i pattern standard di .NET [MM25].

Sebbene la comunicazione sottostante sia gestita semanticamente dallo standard **JSON-RPC 2.0** [ESGK25, ASB25], l'SDK opera una completa astrazione di tale

complessità, privilegiando lo `stdio` (Standard Input/Output) come canale di trasporto primario [ASB25, ESGK25, MM25]. Le funzionalità del server vengono esposte tramite un paradigma dichiarativo basato su attributi. Le classi contenenti la logica dei tool vengono marcate con `[McpServerToolType]`, mentre i singoli metodi che l'LLM dovrà invocare sono decorati con `[McpServerTool]` [MM25]. Per garantire che il modello comprenda correttamente la semantica degli strumenti, è essenziale fornire metadati descrittivi sia ai metodi che ai parametri tramite l'attributo `[Description]` del namespace `System.ComponentModel` [MM25].

L'adozione di questo SDK è giustificata da un'analisi comparativa rispetto all'implementazione manuale del protocollo, dalla quale emerge un netto miglioramento nella *developer experience* e nella sicurezza del codice. Gestire manualmente il protocollo JSON-RPC imporrebbe allo sviluppatore l'obbligo di implementare parser complessi, validare gli schemi di input, gestire il routing delle chiamate e serializzare le risposte, oltre a dover sollevare eccezioni specifiche come `McpErrorCode.InvalidParams` in caso di errori [MM25]. Al contrario, l'approccio gestito dall'SDK automatizza interamente il parsing e la validazione rispetto ai tipi forti di C#, eliminando la necessità di manipolare oggetti non tipizzati e riducendo drasticamente il codice *boilerplate* necessario [MM25]. Procedendo in questo modo si garantisce un *type-safety* superiore e permettendo allo sviluppatore di concentrarsi sullo sviluppo sulla logica di business piuttosto che sull'infrastruttura di comunicazione.

Nel flusso di lavoro del progetto, l'SDK agisce come giuntura tra la logica applicativa C# e le capacità decisionali dell'agente AI. Il processo segue uno schema rigido dove a seguito dell'input utente, l'LLM elabora una richiesta strutturata che viene trasmessa via JSON-RPC [WCP+25, ESGK25]. Il server intercetta tale richiesta, trasforma il payload negli argomenti tipizzati e invoca il metodo C# corrispondente [MM25, WCP+25]. È rilevante notare che l'architettura supporta anche flussi bidirezionali o ricorsivi: l'istanza di `McpServer` può essere utilizzata per richiamare il client (il contesto LLM) per operazioni di *sampling*, come dimostrato da funzioni che necessitano di generare sintesi di contenuti scaricati interagendo nuovamente con il modello [MM25].

2.3 Architettura della Comunicazione

2.3.1 Protocolli di Base: REST, JSON-RPC, WebSocket

L’architettura del sistema si fonda su tre standard di comunicazione distinti ma complementari: REST, JSON-RPC 2.0 e WebSocket. Ognuno di questi protocolli risponde a specifiche esigenze architetturali, dalla gestione delle risorse *stateless* alla chiamata di procedure remote, fino alla comunicazione in tempo reale e la loro cooperazione integrata risulta fondamentale per il corretto funzionamento dell’intera piattaforma.

REST: Lo Standard per le Risorse Web

Il REpresentational State Transfer (**REST**) costituisce lo stile architettonico di riferimento per l’esposizione di interfacce web scalabili e interoperabili [YTY20]. Definito originariamente da Roy Fielding nel 2000, non si configura come una semplice raccolta di pattern, bensì come un insieme organico di vincoli progettuali mirati a ottimizzare scalabilità, efficienza e affidabilità dei sistemi distribuiti [BKA⁺24].

L’architettura si fonda sul protocollo HTTP e sul paradigma di comunicazione stateless: ogni richiesta ha con sé i dati indispensabili alla sua elaborazione, liberando il server dal compito di gestire uno stato persistente per il client [BKA⁺24]. Tale approccio orientato alle risorse identifica ogni entità tramite un URI (*Uniform Resource Identifier*) univoco e ne permette la manipolazione attraverso rappresentazioni standardizzate, tipicamente in formato JSON [YTY20, BKA⁺24].

Un punto importante del modello REST è l’uso dei verbi HTTP con un significato preciso: **GET** serve per leggere in modo sicuro e ripetibile le informazioni, **POST** per aggiungere nuove risorse, e **DELETE** per cancellarle. Usare correttamente questi verbi è fondamentale per garantire che l’interfaccia funzioni nel modo giusto [BKA⁺24]. La diffusione capillare di questo standard, cresciuta esponenzialmente nell’ultimo decennio, è sostenuta anche dalla gestione efficiente del *caching*, che riduce il carico computazionale evitando la rigenerazione di risposte identiche [YTY20].

Nel contesto progettuale, il servizio API esterno adotta questo paradigma per esporre endpoint transazionali quali `POST /muconf/plist` o `/pcreate`. Tuttavia, il modello REST presenta un limite strutturale intrinseco: essendo basato su un ciclo *request-response* sincrono, non supporta nativamente il *server push*, risultando inadeguato per scenari che richiedono aggiornamenti bidirezionali in tempo reale, come le notifiche istantanee [Ces24].

JSON-RPC 2.0: Semplicità nella Chiamata Remota

Per la gestione delle interazioni procedurali complesse, il sistema adotta **JSON-RPC 2.0**, un protocollo *stateless* e leggero specificato nel 2010 per l'invocazione di metodi remoti (RPC) [JSO10]. A differenza di protocolli più complessi come SOAP, JSON-RPC utilizza la sintassi leggera di JSON per definire lo scambio di messaggi, indipendentemente dal livello di trasporto usato (HTTP, socket o stdio) [JSO10].

La struttura del protocollo è rigorosa ma minimale. Una richiesta valida deve includere la versione del protocollo (`"jsonrpc": "2.0"`), il nome del metodo (`method`), gli eventuali parametri (`params`) e un identificativo (`id`) necessario per collegare la risposta [JSO10]. L'omissione dell'identificativo trasforma la chiamata in una *Notification*, per la quale il server non è tenuto a inviare alcuna risposta, ottimizzando il traffico di rete [JSO10, Ces24]. Le risposte, a loro volta, devono restituire un risultato (`result`) o un oggetto di errore strutturato (`error`) contenente codice e messaggio descrittivo [JSO10].

La scelta di JSON-RPC 2.0 è motivata dalla sua capacità di supportare comunicazioni bidirezionali asincrone mantenendo una semplicità implementativa estrema [Ces24]. Nel progetto, esso costituisce la spina dorsale del *Model Context Protocol* (MCP), veicolando comandi critici come `initialize` o `tools/call`. L'utilizzo di un SDK dedicato permette di astrarre interamente le operazioni di *parsing* e serializzazione, garantendo la conformità alla specifica senza dispersionsi compiti per lo sviluppatore [Ces24].

WebSocket: Canale Full-Duplex Real-Time

Mentre REST e JSON-RPC gestiscono efficacemente le interazioni basate su richiesta, l'architettura necessita di un canale dedicato per la trasmissione di eventi

2.3. ARCHITETTURA DELLA COMUNICAZIONE

asincroni a bassa latenza. A tale scopo viene adottato il protocollo **WebSocket**, che permette di instaurare un tunnel di comunicazione **full-duplex** persistente su una singola connessione TCP, superando i limiti strutturali del modello HTTP tradizionale [Ces24].

L'impiego di WebSocket è giustificato dalla necessità di eliminare l'inefficienza delle tecniche di *polling*. Mantenendo il canale aperto, il server può trasmettere dati al client istantaneamente (approccio *push*) senza l'overhead di negoziare una nuova connessione per ogni messaggio. Come evidenziato nella Tabella 2.2, questo approccio riduce drasticamente la latenza e il consumo di banda rispetto alle alternative basate su interrogazione periodica.

Tabella 2.2: Confronto prestazionale: HTTP Polling vs WebSocket.

Metrica	Short Polling	WebSocket
Latenza	Alta (1-5 sec)	Bassissima (<10 ms)
Overhead	Alto (Header HTTP completi)	Minimo (Frame leggeri)
Carico Server	Elevato (Connessioni multiple)	Basso (Connessione persistente)
Scalabilità	Limitata	Elevata

Tuttavia, la gestione "grezza" di un socket persistente introduce complessità infrastrutturali, quali la gestione delle disconnessioni impreviste e l'attraversamento di proxy restrittivi. Per tale motivo, nel progetto l'utilizzo di WebSocket non avviene in forma diretta, ma è mediato dalla libreria **SignalR**, che astrae queste complessità garantendo meccanismi di *fallback* e riconnessione automatica, come verrà dettagliato nel capitolo dedicato alle tecnologie *real-time*.

2.3.2 SignalR: Framework Real-Time

L'evoluzione dei sistemi distribuiti verso architetture reattive impone l'adozione di strumenti capaci di garantire una comunicazione a bassa latenza e alta affidabilità. Per rispondere a tale esigenza, l'architettura del progetto integra **SignalR**, una libreria open-source dell'ecosistema ASP.NET Core progettata per astrarre la complessità della comunicazione bidirezionale tra server e client [RSJI24]. Il framework non segue il tradizionale modello *request-response* di HTTP, abilitando

2.3. ARCHITETTURA DELLA COMUNICAZIONE

un meccanismo di *server push* che permette l’aggiornamento istantaneo delle interfacce client senza ricorrere a tecniche inefficienti di polling [Med22, Mic23].

Architettura e Astrazione del Trasporto. SignalR opera come un livello di astrazione ad alto livello costruito sopra i protocolli di trasporto sottostanti. Il componente fondamentale è il **Transport Layer**, che gestisce la negoziazione della connessione implementando una logica di *fallback* automatico [Rea24]. Il framework tenta inizialmente di stabilire una connessione via **WebSocket**, considerato il trasporto preferenziale per le sue caratteristiche *full-duplex*.

Qualora l’ambiente di rete (proxy, firewall) o le capacità del client non lo supportino, SignalR degrada in modo trasparente verso **Server-Sent Events (SSE)** e, in ultima istanza, verso il **Long Polling** [RSJI24, Mic23]. Il modello di programmazione si basa sul **Hub Pattern**, che funge da pipeline centrale per lo scambio di messaggi. Gli Hub permettono di invocare metodi sul client come se fossero locali, supportando diverse topologie di distribuzione: dal *broadcasting* globale (tramite `Clients.All`) all’indirizzamento *unicast* verso connessioni specifiche (`Clients.Client(id)`) [Rea24, RSJI24]. Per garantire la robustezza del codice, l’architettura sfrutta **Strongly-typed Hubs** basati sull’interfaccia generica `Hub<T>`, che assicurano la verifica dei tipi a tempo di compilazione (*compile-time safety*), prevenendo errori di invocazione comuni nei sistemi basati su stringhe magiche [RSJI24]. Un aspetto critico per la resilienza del sistema è la gestione del ciclo di vita della connessione. SignalR espone hook di eventi quali `OnConnectedAsync` e `OnDisconnectedAsync` per la gestione delle risorse e implementa una politica di **riconnessione automatica** configurabile. Nel progetto, è stata adottata una strategia di *retry* con backoff incrementale a 0, 2, 5 e 10 secondi, permettendo il recupero trasparente della sessione in caso di instabilità transitoria della rete [Rea24].

Vantaggi Operativi e Produttività. La scelta di adottare SignalR in luogo di un’implementazione diretta di WebSocket (*raw WebSocket*) è giustificata da un’analisi costi-benefici legata alla produttività e alla manutenibilità. Mentre WebSocket fornisce un canale di comunicazione grezzo, il suo utilizzo diretto imporrebbe agli sviluppatori di re-implementare funzionalità infrastrutturali complesse. SignalR riduce drasticamente il codice *boilerplate* necessario, stimato in una riduzione del

2.3. ARCHITETTURA DELLA COMUNICAZIONE

90%, gestendo internamente la serializzazione dei messaggi, il routing e la gestione degli errori [RSJI24]. La Tabella 2.3 evidenzia le differenze strutturali tra le due tecnologie, sottolineando come SignalR risolva nativamente le criticità legate alla gestione manuale del protocollo.

Tabella 2.3: Confronto tecnico: WebSocket Raw vs SignalR.

Aspetto	WebSocket Raw	SignalR
Fallback	Manuale	Automatico (SSE → LP)
Riconnessione	Custom (logica complessa)	Automatica con backoff
Messaggistica	Serializzazione manuale	Metodi Strongly-typed
Broadcasting	Loop manuale sulle socket	<code>Clients.All.Method()</code>
Complessità	Elevata (300-500 righe)	Minima (20-30 righe)

Oltre all'efficienza del codice, l'adozione del framework migliora significativamente la *Developer Experience* (DX) grazie all'integrazione con IntelliSense e alla sicurezza dei tipi offerta dall'ecosistema .NET, riducendo la superficie di errore durante lo sviluppo di funzionalità real-time [Rea24].

Ruolo nell'Architettura di Progetto: SignalRHub. Nell'architettura del progetto di tesi, SignalR non è un semplice accessorio ma costituisce la colonna portante per l'interattività. Il componente centrale è identificato come **SignalRHub**, un Hub esterno preesistente ospitato sul server Kestrel (porta 7193), configurato con autenticazione JWT e policy CORS permissiva per accettare connessioni dai client distribuiti [RSJI24].

Il sistema implementa un **Dual-Channel Pattern**, separando logicamente le responsabilità di comunicazione:

1. **Canale REST**: Utilizzato per operazioni CRUD pesanti e transazionali.
2. **Canale SignalR**: Dedicato a notifiche leggere e aggiornamenti di stato in tempo reale.

Questa separazione garantisce la resilienza del sistema e una chiara *separation of concerns*. SignalRHub agisce da ponte (*bridging*) tra due componenti principali:

1. **SignalR Client .NET (Lato MCP Server)**: Il server MCP utilizza il pacchetto `Microsoft.AspNetCore.SignalR.Client` per stabilire una connessione

persistente con l'Hub tramite il metodo `StartAsync()`. Il client sottoscrive eventi specifici tramite `On<T>` per reagire ai comandi e utilizza `InvokeAsync` per inviare risultati o notifiche di stato, sfruttando la politica di riconnessione automatica (0-10s) per garantire la continuità operativa anche in caso di riavvio dell'Hub.

2. SignalR Client Browser (Lato Configurator3D): Il frontend, basato sul pacchetto npm `@microsoft/signalr` v8.x, gestisce la visualizzazione dinamica. I gestori di eventi, come `on("open")` o `on("create")`, intercettano i comandi provenienti dal server MCP. Per garantire la coerenza distribuita, è stato implementato un **ACK pattern** indispensabile per la corretta elaborazione di un comando. Il client invoca il metodo `invoke("SendAck", commandId)`, confermando l'avvenuta ricezione ed esecuzione dell'operazione.

Caratteristiche Determinanti per la Scelta. I fattori che hanno guidato la decisione di integrare SignalR nell'architettura del progetto sono più di uno e rispondono a requisiti non funzionali critici:

- **Bidirezionalità Real-Time:** La capacità di mantenere una latenza inferiore ai 10ms, contro gli 1-5 secondi tipici del polling, è essenziale per la fluidità del configuratore 3D.
- **Affidabilità:** La gestione automatica delle riconnessioni assicura la stabilità delle sessioni utente di lunga durata.
- **Fallback Transport:** La garanzia di compatibilità al 100% in ambienti di rete restrittivi elimina i rischi di disservizio lato client.
- **Maturità dell'Ecosistema:** Il supporto LTS di Microsoft e la natura *production-ready* della libreria offrono garanzie di manutenibilità a lungo termine [Mic23, RSJI24].

2.4 Sicurezza Applicativa

2.4.1 JWT: Autenticazione Stateless

La sicurezza delle moderne architetture a microservizi richiede meccanismi di autenticazione che garantiscano scalabilità e disaccoppiamento. Per questo lavoro si è

2.4. SICUREZZA APPLICATIVA

adotta il formato JSON Web Token (JWT), definito dalla RFC 7519 [JBS15], per rappresentare in modo compatto e sicuro le asserzioni (claims) scambiate tra le parti comunicanti. Dal punto di vista strutturale, il token si presenta come una stringa codificata in Base64Url, suddivisa in tre segmenti distinti separati da un punto, i componenti sono: *Header*, *Payload* e *Signature*. L'**Header** definisce i metadati crittografici, specificando l'algoritmo di firma (es. `{"alg": "HS256", "typ": "JWT"}`), mentre il **Payload** incapsula le asserzioni vere e proprie. Queste includono claim standard registrati come il soggetto (`sub`), la scadenza (`exp`) e dati applicativi personalizzati, rendendo il token un oggetto informativo autosufficiente. La **Signature**, infine, è ottenuta applicando l'algoritmo specificato (come HMAC-SHA256) alla concatenazione di header e payload codificati, utilizzando una chiave segreta nota solo al server. Questo meccanismo permette al ricevente di verificare l'integrità del dato e l'autenticità dell'emittente ricalcolando la firma al momento della ricezione [JBS15, Shi18].

La decisione di adottare JWT invece delle sessioni server-side è motivata da chiari vantaggi per i sistemi distribuiti infatti essendo stateless, i JWT rimuovono la necessità di sincronizzare lo stato delle sessioni tra nodi o di interrogare un database centralizzato ad ogni richiesta HTTP, favorendo una scalabilità orizzontale più semplice (vedi Tabella 2.4) [LJK17, OFMWY17].

Tabella 2.4: Confronto: Session-based Auth vs JWT.

Aspetto	Session-based	JWT
Stato	Stateful (Server-side)	Stateless (Client-side)
Storage	Memoria Server / Redis	Client (LocalStorage/Cookie)
Scalabilità	Limitata (Sticky Sessions)	Elevata (Horizontal Scaling)
CSRF	Vulnerabile	Generalmente Immune
Cross-domain	Problematico	Semplice (CORS-friendly)

Essendo **self-contained**, il token trasporta già tutte le informazioni necessarie per l'autorizzazione, riducendo la latenza di rete e il carico sui database. Inoltre, la natura standardizzata del formato JSON facilita l'interoperabilità *cross-domain*, superando le restrizioni tipiche dei cookie in scenari CORS (*Cross-Origin Resource Sharing*) complessi [OFMWY17].

Il flusso di autenticazione prevede che il **servizio API esterno** agisca come *Identity Provider*. Dopo una richiesta di login valida (`POST /api/login`), il server emette un JWT firmato contenente i claim dell’utente. Il client, rappresentato dall’**MCP Server**, riceve il token e lo include nell’header `Authorization` di tutte le richieste successive, usando lo schema standard `Bearer <token>`. Alla ricezione, il servizio API esterno verifica soltanto la firma crittografica e il campo di scadenza (`exp`), senza consultare tabelle di sessione. Per ridurre i rischi legati alla memorizzazione lato client, il sistema richiede l’uso esclusivo di canali cifrati HTTPS e adotta finestre di validità brevi (15–30 minuti), minimizzando l’impatto di un eventuale furto del token [Shi18, JBS15].

2.4.2 CORS e Transport Security

Quando frontend e backend risiedono su domini o porte diverse, è necessario abilitare Cross-Origin Resource Sharing (CORS) per il controllo degli accessi e utilizzare TLS per cifrare le comunicazioni tra il client di configurazione 3D e l’hub di backend.

La sicurezza dei browser impone nativamente la *Same-Origin Policy* (SOP), una misura difensiva che impedisce a uno script caricato da un’origine (definita dalla combinazione di protocollo, dominio e porta) di interagire con risorse provenienti da un’origine diversa [Moz]. Tale restrizione, sebbene fondamentale per prevenire attacchi di tipo CSRF (*Cross-Site Request Forgery*), rappresenta un ostacolo tecnico nello scenario implementativo del progetto. Il frontend dell’applicazione è ospitato su un sottodomino dedicato (es. `https://app.dominio.com`), mentre il backend, che espone gli endpoint REST e l’hub SignalR, risponde su un indirizzo differente o su una porta specifica (es. `https://api.dominio.com:7193`). Essendo differenti l’origine e la porta di comunicazione, il browser classifica le richieste come *cross-origin*, bloccando di default l’instaurazione della connessione WebSocket necessaria per la comunicazione in tempo reale.

Per far sì che questa limitazione non impatti eccessivamente la sicurezza dell’applicazione, è stato configurato il middleware CORS lato server. Il protocollo CORS consente al backend di dichiarare esplicitamente quali origini sono autorizzate ad accedere alle risorse tramite l’invio di specifici header HTTP [W3C]. Quando il

2.4. SICUREZZA APPLICATIVA

browser rileva una richiesta verso un dominio esterno che potrebbe comportare effetti collaterali sui dati utente, avvia in modo precauzionale una "Preflight request" utilizzando il metodo HTTP **OPTIONS**. Come illustrato nel Diagramma 2.1, il server risponde a questa richiesta preliminare specificando i metodi consentiti e, crucialmente, l'header **Access-Control-Allow-Origin**.

2.4. SICUREZZA APPLICATIVA

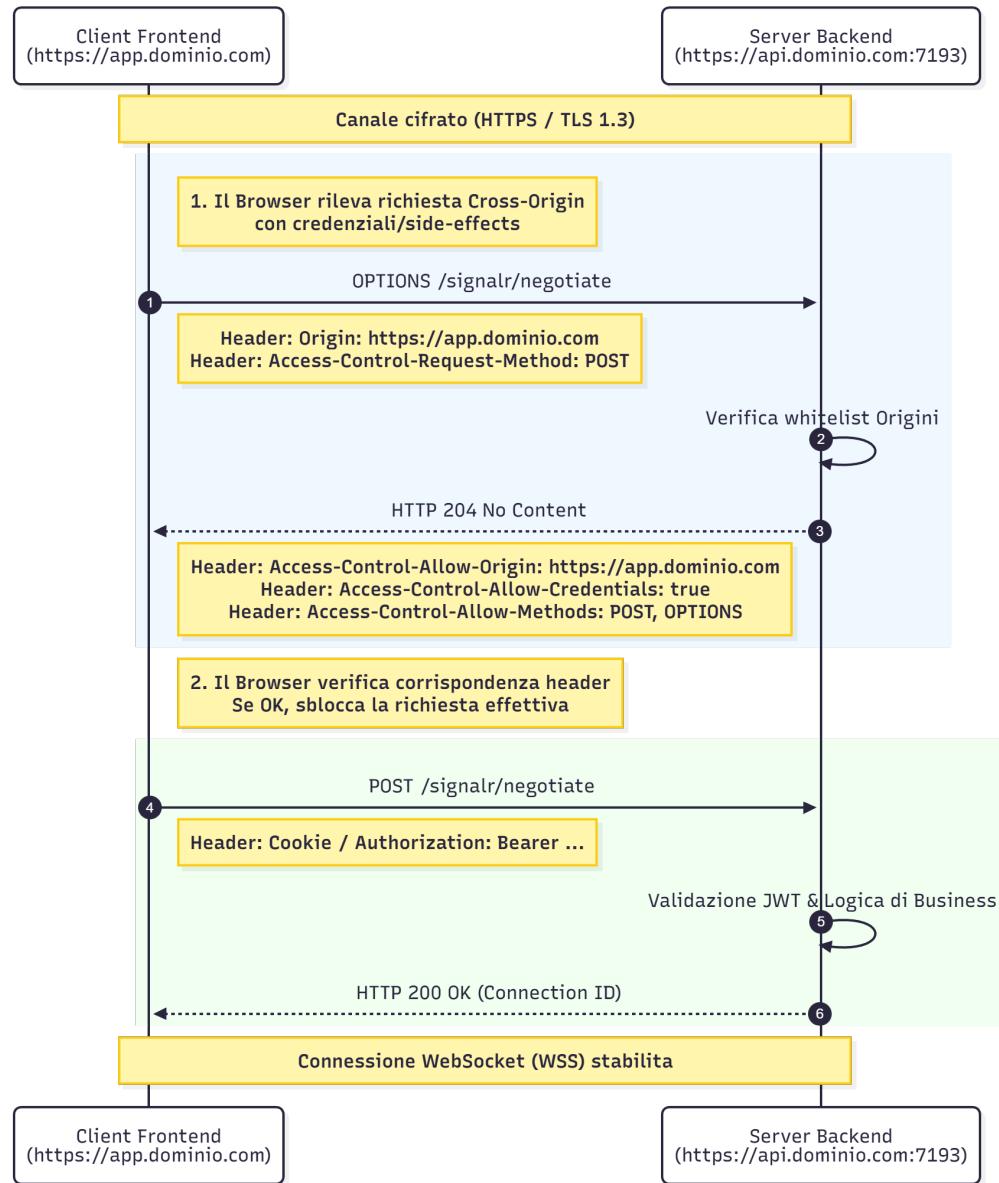


Figura 2.1: Diagramma di sequenza del flusso CORS Preflight tra Frontend e Backend. Il server autorizza esplicitamente l'origine e l'uso delle credenziali prima della connessione WebSocket.

Nel caso specifico della comunicazione real-time con SignalR, la configurazione

si complica maggiormente in quanto vi è la necessità di trasmettere credenziali di autenticazione (cookie o header di autorizzazione). La specifica CORS impone che, quando le credenziali sono coinvolte, il server non possa utilizzare il carattere jolly (*) per l'origine, ma debba restituire l'esatto dominio del chiamante e includere l'header `Access-Control-Allow-Credentials: true` [Moz]. L'assenza di tale configurazione provocherebbe il fallimento immediato dell'handshake WebSocket inizializzato dal client.

Parallelamente al controllo degli accessi, la confidenzialità e l'integrità dei dati in transito sono garantite dall'adozione del protocollo *Transport Layer Security* (TLS), su cui si basano HTTPS (*HyperText Transfer Protocol Secure*) e WSS (*WebSocket Secure*). L'intero traffico tra il configuratore 3D e le API di backend avviene esclusivamente su canale cifrato, utilizzando prevalentemente lo standard TLS 1.3. La nuova versione del protocollo riduce la latenza durante l'handshake e rimuove algoritmi crittografici obsoleti, limitando la possibilità di sorveglianza del traffico [dCdCvO22].

L'adozione di HTTPS è imprescindibile non solo per le API REST, ma soprattutto per proteggere il token di autenticazione JSON Web Token (JWT), impedendo la sua intercettazione. In assenza di un canale cifrato, un attaccante posizionato sulla rete (attacco *Man-In-The-Middle*) potrebbe intercettare il token in chiaro e impersonare l'utente. Il protocollo TLS stabilisce una crittografia end-to-end che assicura che i dati scambiati siano leggibili solo dagli endpoint autorizzati, garantendo contemporaneamente l'autenticità del server tramite certificati digitali [SMF⁺19]. Dal punto di vista crittografico, il canale utilizza cifrari simmetrici moderni come AES-256-GCM (*Galois/Counter Mode*), che forniscono cifratura autenticata garantendo sia la segretezza che l'integrità del messaggio contro tentativi di manomissione [dCdCvO22].

2.5 Integrazione Frontend

2.5.1 TypeScript: Type-Safety JavaScript

Definizione e Paradigma del Linguaggio

Nel panorama dello sviluppo web moderno, **TypeScript** rappresenta un’evoluzione strutturale del tradizionale scripting lato client. Definito formalmente come un superset sintattico di JavaScript, questo linguaggio, sviluppato e mantenuto da Microsoft come progetto open-source, integra un sistema di tipizzazione statica opzionale all’interno della flessibilità dinamica propria dello standard ECMAScript [Mic24c, TC324].

L’architettura del linguaggio, giunta alla versione 5.x, non è destinata all’esecuzione diretta nei browser o nei runtime come Node.js. Essa necessita invece di un processo di compilazione, o più propriamente detto di *transpilation*, gestito dal compilatore `tsc`. Il processo di compilazione converte TypeScript in JavaScript conforme a ECMA-262, assicurando compatibilità cross-platform e rimuovendo le annotazioni di tipo, che sono utili solo in fase di sviluppo [Mic24c, Mic24d].

Analisi Comparativa: Static vs Dynamic Typing

L’adozione di TypeScript risponde alle criticità dello sviluppo su larga scala in JavaScript: grazie ai tipi statici opzionali, diminuisce la probabilità di errori, facilita il refactoring e migliora la manutenibilità del codice. Come evidenziato dalla documentazione tecnica relativa alla tipizzazione statica, la verifica dei vincoli sui dati in JavaScript avviene esclusivamente a *runtime*, esponendo il software a errori che emergono solo durante l’esecuzione [Moz24]. Al contrario, TypeScript anticipa tale validazione alla fase di compilazione (*compile-time*), permettendo l’identificazione preventiva di incongruenze logiche e sintattiche [Mic24c].

La Tabella 2.5 riassume le differenze strutturali tra i due approcci, evidenziando l’impatto sulla stabilità del codice e sull’esperienza di sviluppo.

Un vantaggio determinante derivante dall’adozione di questo paradigma è il potenziamento degli strumenti di supporto allo sviluppo. Ambienti di sviluppo integrati come Visual Studio Code sfruttano le definizioni di tipo per offrire funzionalità avanzate di **IntelliSense**, quali il completamento automatico del codice,

Tabella 2.5: Confronto architetturale: JavaScript vs TypeScript.

Aspetto Funzionale	JavaScript (Standard ECMA)	TypeScript
Verifica dei Tipi	Runtime (Dinamica)	Compile-time (Statica)
Rilevamento Errori	In Produzione/Esecuzione	In Sviluppo/Compilazione
Supporto Tooling	IntelliSense Limitato	Autocomplete Completo
Refactoring	Rischioso (String-based)	Sicuro (Type-checked)

la documentazione in linea e la navigazione semantica delle API [Mic24b]. Tale caratteristica permette di diminuire notevolmente il rischio di errori banali, come refusi nei nomi dei metodi o accessi a proprietà inesistenti, che in un contesto puramente JavaScript verrebbero intercettati solo a runtime [Mic24c, Mic24b].

Implementazione nel Contesto Progettuale

Nell'architettura del configuratore 3D, l'impiego di TypeScript (versione 5.x) costituisce la base per garantire la robustezza del frontend. La necessità di manipolare strutture dati complesse per la configurazione tridimensionale e la gestione di comunicazioni real-time richiede garanzie di correttezza che trascendono le capacità del JavaScript puro.

In particolare, l'integrazione del client **SignalR** avviene attraverso il pacchetto `@microsoft/signalr`, il quale fornisce definizioni di tipo native [Mic24a]. Questo approccio permette di strutturare le invocazioni dei metodi remoti e la gestione degli eventi in modo *type-safe*. Ogni messaggio scambiato tra client e server viene validato rispetto a interfacce predefinite, assicurando che il payload dei dati rispetti rigorosamente la struttura attesa dall'Hub. L'adozione di TypeScript rende le API di comunicazione verificabili staticamente, riducendo la fragilità tipica dei contesti dinamici e aumentando manutenibilità e scalabilità del client [Mic24c, Mic24a].

2.5.2 SignalR Client Browser

Il Package `@microsoft/signalr`

L'implementazione lato client della comunicazione real-time nel progetto *Configurator3D* si basa sul package `@microsoft/signalr`, una libreria JavaScript ufficiale distribuita attraverso il registro npm e progettata per fornire un'interfaccia completa

e type-safe per l’interazione con hub SignalR remoti [Mic24a]. Questa componente software, allineata alla versione 8.x del framework server-side, costituisce il layer di astrazione che permette alle applicazioni browser-based e ai runtime Node.js di stabilire connessioni bidirezionali con backend ASP.NET Core, gestendo automaticamente la negoziazione del protocollo di trasporto e la serializzazione dei messaggi [Mic23].

Dal punto di vista architetturale, il package espone un insieme di API object-oriented incentrate sulla classe `HubConnectionBuilder`, che implementa il pattern *Builder* per la configurazione dichiarativa delle connessioni. L’oggetto `HubConnection` risultante rappresenta il canale di comunicazione attraverso cui avviene lo scambio di messaggi con il server. Le primitive fondamentali dell’API adottano una semantica asincrona basata su Promise, assicurando compatibilità nativa con il modello di concorrenza moderno di JavaScript. Nello specifico, il metodo `start()` inizializza la connessione effettuando l’handshake con il server, mentre `on(eventName, handler)` permette la registrazione di callback per eventi specifici inviati dal server, realizzando un pattern *Observer* distribuito [Mic24a, Mic25a].

La capacità di invocare metodi remoti in modo programmatico è fornita dalla primitiva `invoke(methodName, ...args)`, che serializza i parametri, li trasmette al server attraverso il canale WebSocket (o il trasporto di fallback attivo), e restituisce una Promise risolta con il valore di ritorno del metodo server-side. Questo meccanismo di *Remote Procedure Call* (RPC) permette di ridurre significativamente la complessità dello sviluppo di interfacce distribuite, nascondendo i dettagli del protocollo sottostante e della gestione degli errori di rete [Mic23].

Configurazione e Resilienza della Connessione

La robustezza della comunicazione real-time in ambienti di produzione richiede una configurazione precisa dei meccanismi di resilienza e diagnostica. Il `HubConnectionBuilder` mette a disposizione metodi fluent per definire i parametri operativi critici della connessione, offrendo la flessibilità necessaria ad adattarsi a diverse topologie di rete e requisiti di affidabilità [Mic23].

Il metodo `withUrl(url, options)` definisce l’endpoint del servizio SignalR e, opzionalmente, configura aspetti avanzati del trasporto, come l’inclusione di

header HTTP personalizzati o la specificazione esplicita dei protocolli di trasporto ammissibili. La resistenza alle disconnessioni temporanee, fenomeno impossibile da evitare del tutto in reti mobili o congestionate, è gestita dalla direttiva `withAutomaticReconnect(retryDelays)`, che implementa una strategia di riconnessione automatica con backoff esponenziale. La specifica dei ritardi di retry, tipicamente configurati come array crescente (ad esempio [0, 2000, 5000, 10000] millisecondi), consente di bilanciare la reattività nel ripristino della connessione con la necessità di evitare sovraccarichi sul server in scenari di degradazione estesa del servizio [Mic23, Goo22a].

Il monitoraggio diagnostico dello stato della connessione è facilitato dal metodo `configureLogging(level)`, che attiva la tracciatura dettagliata degli eventi del ciclo di vita della connessione, inclusi handshake, fallimenti di trasporto e messaggi scambiati. La definizione di handler per eventi specifici avviene per mezzo delle invocazioni del metodo `on(eventName, callback)`, dove `eventName` corrisponde al nome del metodo definito nell’interfaccia strongly-typed dell’hub server. Nel contesto progettuale, eventi quali "GetMessage" e "GetConnectionId" sono intercettati per aggiornare lo stato dell’applicazione client in risposta a notifiche push dal backend [Mic24a].

La invocazione di metodi sul server avviene tramite la primitiva `invoke(methodName, ...args)`, che serializza automaticamente i parametri tipizzati in formato JSON. Ad esempio, `invoke("SendMessage", receiverId, message)` invia al server una richiesta di inoltro di un messaggio a un destinatario specifico. La chiamata restituisce una `Promise` che si risolve al completamento dell’operazione o viene respinta in caso di errore di rete o applicativo. Il ciclo di vita della connessione è gestito dai metodi `start()`, che apre il canale asincrono, e `stop()`, che chiude la connessione in modo ordinato; inoltre il callback `onclose(error)` viene invocato quando si verifica una disconnessione, sia intenzionale sia dovuta a un errore [Mic23, Mic25a].

Integrazione Applicativa e Pattern Operativi

Nel contesto dell’architettura del configuratore tridimensionale, il client SignalR assume il ruolo di ponte da comunicazione tra l’interfaccia utente e il server MCP

(*Model Context Protocol*), gestendo lo scambio di comandi e notifiche di stato. La registrazione di handler per eventi specifici, corrispondenti ai comandi del protocollo applicativo, è implementata mediante invocazioni successive del metodo `on()`. Comandi quali "open", "create" e "add" sono intercettati da callback che aggiornano atomicamente lo stato del modello 3D visualizzato, garantendo coerenza tra la rappresentazione locale e lo stato remoto gestito dal backend [RSJI24].

Per assicurare l'affidabilità delle operazioni distribuite, è stato adottato un pattern di acknowledgment esplicito. Al completamento dell'elaborazione di un comando ricevuto, il client invoca il metodo `invoke("SendAck", commandId, status)`, trasmettendo al server la conferma dell'avvenuta ricezione ed esecuzione. Questo schema, ispirato ai protocolli di comunicazione affidabile, permette al server di rilevare eventuali perdite di messaggi o timeout di elaborazione. Attivando strategie di compensazione, come la ritrasmissione o la notifica di errore all'utente, si mitiga l'impatto dei guasti temporanei [RSJI24, Med22].

Capitolo 3

Progettazione del Sistema

3.1 Architettura Generale del Sistema

La progettazione del sistema si fonda su un'architettura definita a strati (*Layered Architecture*), realizzata per garantire una separazione delle responsabilità tra i componenti logici e per favorire la manutenibilità e la testabilità del software. Il design deve armonizzare tre linguaggi: quello umano (IA), quello tecnico (gestionali) e quello visivo (3D).

L'architettura proposta agisce come un intermediario intelligente, lavorando per disaccoppiare la logica decisionale dell'AI dai dettagli operativi dei sottosistemi sottostanti. Come viene illustrato nel Diagramma 1, il sistema comprende tre livelli funzionali principali: il livello di definizione degli strumenti MCP, il livello di servizio per la comunicazione in tempo reale e il livello di integrazione con i sistemi esterni.

3.1. ARCHITETTURA GENERALE DEL SISTEMA

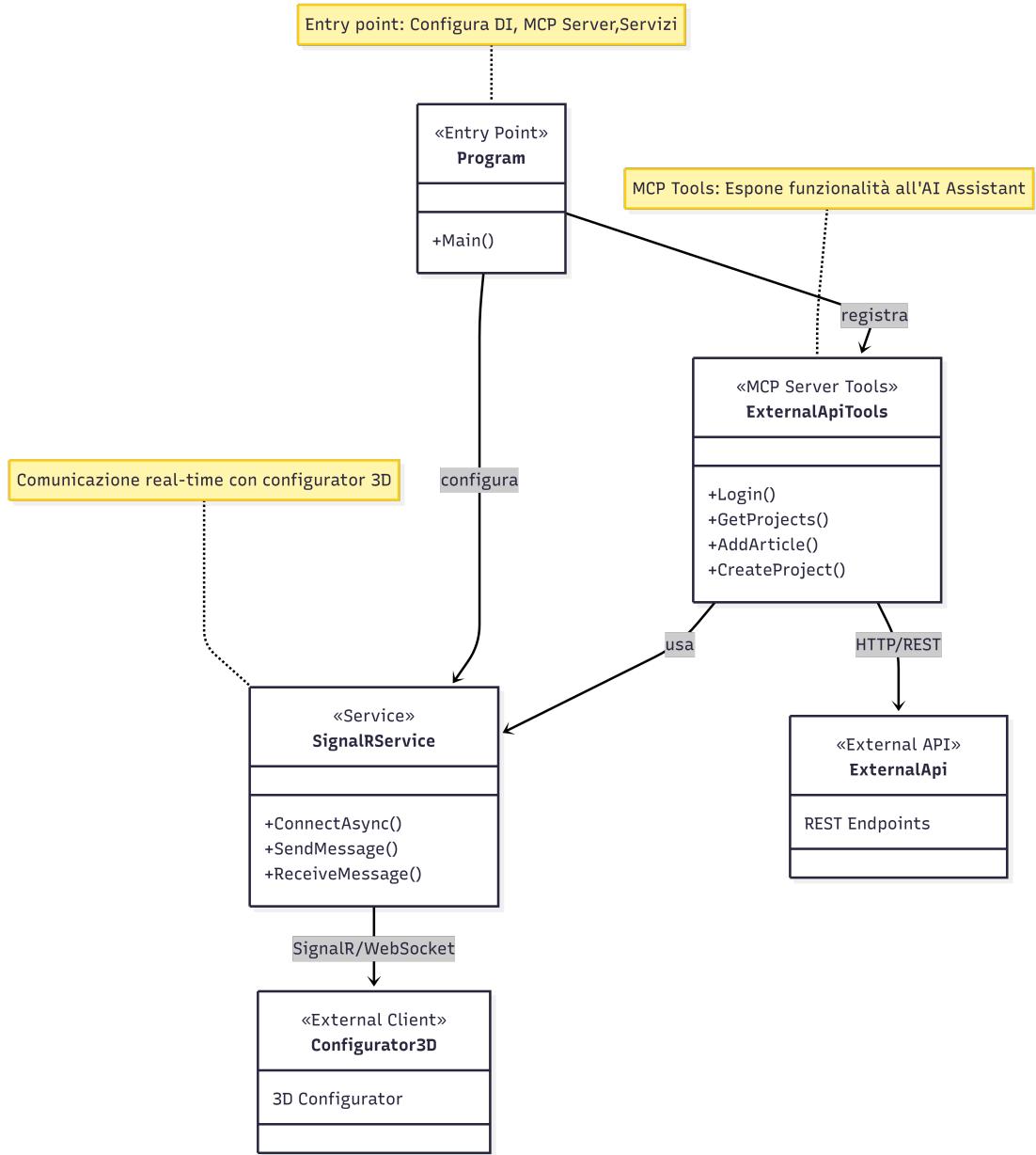


Figura 3.1: Diagramma 1: Architettura di alto livello: componenti principali e relazioni tra Program, ExternalApiTools, SignalRService, ExternalApi e Configurator3D.

Il Pattern Architetturale e i Componenti Logici

Al vertice della gerarchia logica si colloca il modulo dedicato agli strumenti MCP (identificato nel diagramma come `ExternalApiTools`). La loro responsabilità è esporre all'AI Assistant un set di funzionalità astratte, come l'autenticazione, la gestione dei progetti o l'inserimento di articoli, mascherando la complessità delle chiamate sottostanti. Questo livello opera come un traduttore che è in grado di convertire i comandi, espressi in linguaggio umano, in comandi eseguibili dal sistema.

A supporto delle operazioni che richiedono un'interazione immediata con l'interfaccia utente, interviene il livello di servizio real-time, rappresentato dalla classe `SignalRService`. Questo modulo implementa il pattern *Bridge*, agendo come canale di comunicazione asincrono verso il configuratore grafico (`Configurator3D`). La scelta di segregare la logica di comunicazione in un servizio dedicato permette di isolare il protocollo di trasporto dal resto della logica di business, facilitando eventuali evoluzioni future dell'infrastruttura di messaggistica senza impattare sulla definizione dei tool MCP.

I sistemi esterni, sebbene fisicamente separati dal server MCP, costituiscono parte integrante dell'ecosistema progettato. Da un lato, le API del sistema gestionale (rappresentate dalla classe `ExternalApi`) forniscono la persistenza dei dati e la logica di business core attraverso un'interfaccia REST; dall'altro, il client grafico (`Configurator3D`) agisce come terminale di visualizzazione, ricevendo comandi di aggiornamento della scena e restituendo feedback sullo stato della configurazione.

Strategie di Comunicazione e Protocolli

Un aspetto distintivo dell'architettura risiede nella natura ibrida dei pattern di comunicazione adottati. Il sistema adotta due paradigmi differenti che sono:

1. **Comunicazione Sincrona (HTTP/REST):** Utilizzata per l'interazione tra il livello dei tool MCP e le API del gestionale esterno. Questo approccio è stato scelto per operazioni atomiche e transazionali, come il login o il recupero di liste di progetti, dove la consistenza immediata della risposta ha priorità massima.

2. **Comunicazione Asincrona (SignalR/WebSocket):** Adottata per il dialogo tra il server MCP e il client 3D. La necessità di modificare la scena grafica in tempo reale, senza imporre al client un dispensioso polling continuo, ha guidato la scelta verso un protocollo *full-duplex* capace di inviare notifiche push.

Gestione delle Dipendenze e Configurazione

Il punto di ingresso dell'applicazione, identificato dalla classe `Program`, oltre all'avvio del progetto, ha il ruolo di *Composition Root* ossia configura il contenitore di *Dependency Injection* (DI), responsabile dell'istanziazione e del ciclo di vita di tutti i servizi.

L'integrazione della Dependency Injection deriva dalla necessità di invertire il controllo tra i componenti del sistema. La classe `ExternalApiTools` non deve generare al proprio interno le istanze di `SignalRService` o dei client HTTP, ma ne riceve le astrazioni tramite interfacce al momento della sua creazione.

Dunque, l'architettura è strutturata per adattarsi agilmente ai cambiamenti dei sistemi terzi. Il server MCP funge da adattatore che normalizza i flussi di dati, offrendo all'Intelligenza Artificiale un punto di accesso coerente e delegando ai servizi sottostanti l'intera gestione dei protocoli di comunicazione.

3.2 Sistema di Configurazione e Bootstrap

La fase di inizializzazione, o *bootstrap*, costituisce il momento fondante del ciclo di vita applicativo, durante il quale vengono stabilite le condizioni operative necessarie per la corretta esecuzione del server MCP. La progettazione di questo stadio definisce una strategia strutturata per la gestione delle dipendenze, l'isolamento dei parametri di configurazione e la predisposizione dei canali di comunicazione sicuri. L'architettura adotta un approccio dichiarativo, delegando al componente `Program` (l'*entry point*) la responsabilità di orchestrare il caricamento delle impostazioni e la registrazione dei servizi nel contenitore di *Dependency Injection* (DI).

3.2. SISTEMA DI CONFIGURAZIONE E BOOTSTRAP

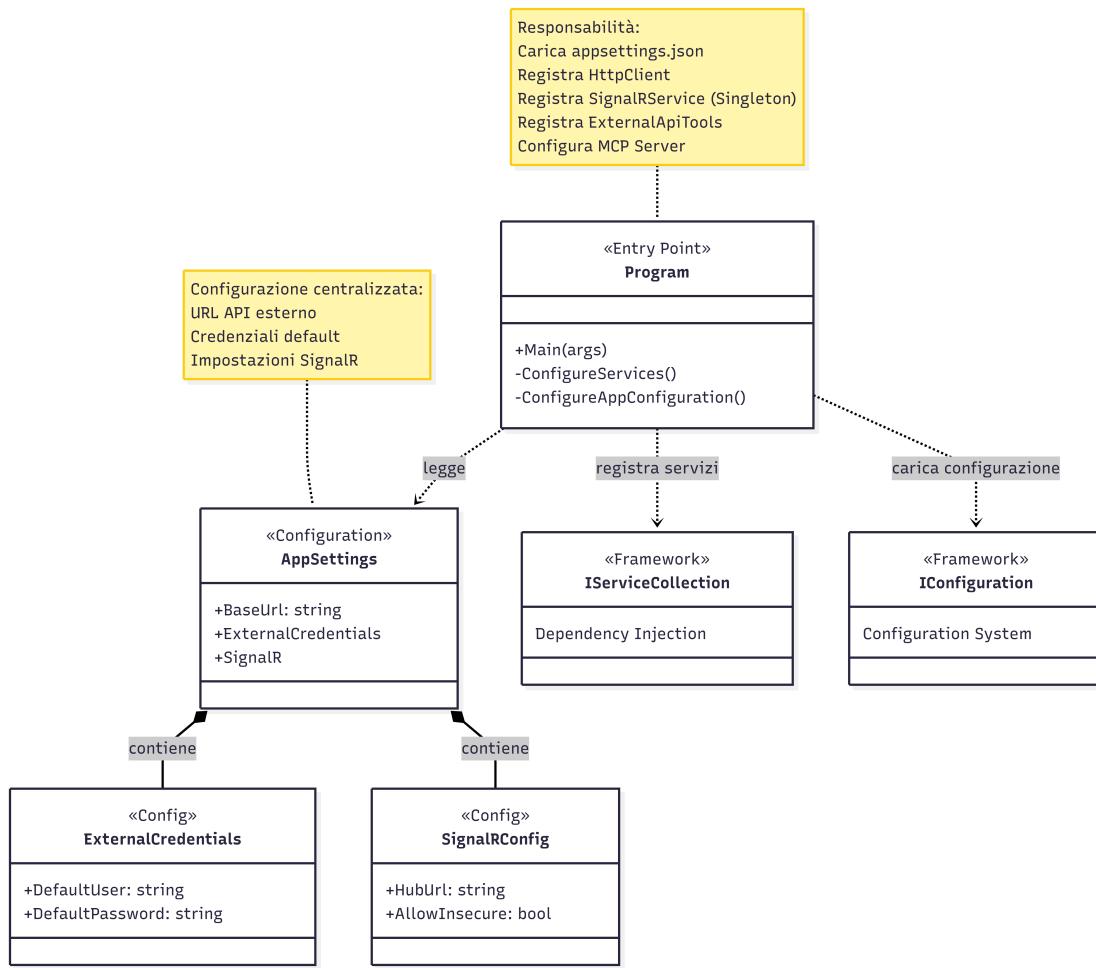


Figura 3.2: Diagramma 2: Layer di configurazione: struttura di **AppSettings** (con **ExternalCredentials** e **SignalRConfig**) e relazioni con **Program**, **IServiceCollection** e **IConfiguration**.

Modellazione e Gestione della Configurazione

Un principio cardine adottato nel design del sistema è la completa esternalizzazione della configurazione, eliminando qualsiasi forma di codifica rigida (*hardcoding*) di parametri sensibili o variabili d'ambiente all'interno del codice sorgente. Come illustrato nel Diagramma 2, il modello di configurazione è stato strutturato gerarchi-

camente attraverso la classe `AppSettings`, che funge da radice per l'aggregazione di tutte le impostazioni applicative.

Questa classe mappa direttamente la struttura del file `appsettings.json`, suddividendo le responsabilità in sotto-moduli specializzati:

- **ExternalCredentials**: incapsula le credenziali di accesso predefinite (utente e password) necessarie per l'autenticazione iniziale verso le API del sistema gestionale esterno.
- **SignalRConfig**: definisce i parametri di connessione per il bridge real-time, inclusi l'URL dell'Hub e i flag di sicurezza per la gestione dei certificati SSL in ambienti di sviluppo (`AllowInsecure`).
- **BaseUrl**: specifica l'endpoint radice per le chiamate REST, permettendo di variare l'ambiente di destinazione senza ricompilare l'applicazione.

Il caricamento di tali parametri è gestito dall'interfaccia `IConfiguration`, che implementa una strategia a livelli di priorità. Il sistema è progettato per supportare configurazioni specifiche per l'ambiente (*environment-specific settings*), caricando in sequenza il file di base e successivamente le sovrascritture specifiche (es. `appsettings.Development.json` o `appsettings.Production.json`). Questo approccio permette di escludere i file contenenti credenziali reali dal sistema di controllo versione (*version control*), mantenendo i segreti confidenziali.

Orchestrazione dei Servizi e Dependency Injection

Una volta caricata la configurazione, il flusso di bootstrap procede con la registrazione dei servizi nel framework. Questa fase serve a definire il ciclo di vita degli oggetti e come essi interagiscono tra loro. Il design sfrutta il pattern della *Dependency Injection* per disaccoppiare l'implementazione dei servizi dalla loro utilizzazione nei tool MCP.

Particolare attenzione è stata posta nella definizione del ciclo di vita del `SignalRService`. A differenza dei componenti stateless, questo servizio gestisce una connessione WebSocket persistente verso il client 3D. Di conseguenza, è stato configurato come **Singleton** che crea un'istanza condivisa all'avvio dell'applicazione e mantenuta viva per l'intera durata dell'esecuzione. Questa scelta assicura

3.3. SERVIZIO DI COMUNICAZIONE REAL-TIME

che tutti i tool MCP che necessitano di inviare aggiornamenti real-time utilizzino il medesimo canale di comunicazione, evitando la proliferazione di connessioni ridondanti e garantendo la coerenza dello stato di connessione.

Parallelamente, vengono registrati i servizi per la comunicazione HTTP e il nucleo stesso del server MCP ([ExternalApiTools](#)). Il metodo `ConfigureServices` agisce quindi come punto di convergenza dove le configurazioni statiche, caricate dagli `AppSettings`, vengono iniettate nei servizi dinamici, rendendo il sistema pronto a gestire le richieste dell'agente AI in modo robusto e configurabile.

3.3 Servizio di Comunicazione Real-Time

L'orchestrazione della comunicazione bidirezionale tra il server MCP e il client grafico è affidata al componente `SignalRService`. Tale classe non è stata concepita come un mero involucro (*wrapper*) delle librerie di trasporto, bensì come un servizio infrastrutturale incaricato di isolare la logica di dominio dalle specifiche del protocollo WebSocket, garantendo la gestione del canale di comunicazione in modo centralizzato.

Gestione del Ciclo di Vita e Connessione

Il `SignalRService` si interpone come uno strato di astrazione tra i moduli applicativi e l'oggetto `HubConnection`. Il compito principale del servizio è la gestione del ciclo di vita della connessione, encapsulata nella procedura interna `TryBuildAndStartConnectionAsync`.

Questa logica implementa meccanismi di riconnessione automatica e politiche di *retry*, fondamentali per assicurare che il bridge tra MCP e configuratore rimanga attivo senza interventi manuali in caso di micro-interruzioni di rete. Dal punto di vista della sicurezza, il servizio è configurato per gestire la negoziazione dei trasporti e, in fase di sviluppo, supporta l'interazione con certificati *self-signed*, facilitando il testing locale del sistema.

3.3. SERVIZIO DI COMUNICAZIONE REAL-TIME

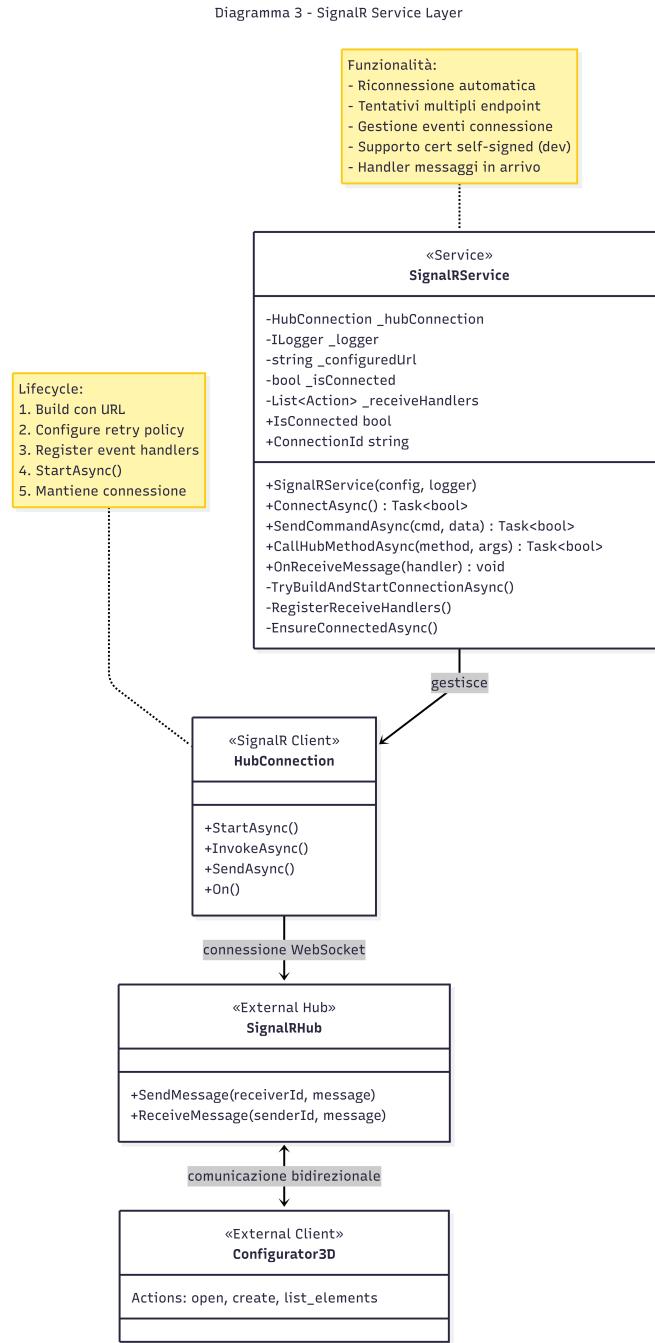


Figura 3.3: Diagramma delle classi del SignalR Service Layer: astrazione della connessione e interfacce verso l'Hub.

Primitive di Comunicazione e Flussi Dati

L’interfaccia pubblica del servizio è stata modellata per riflettere le necessità operative del sistema, fornendo metodi specializzati per l’invio e la ricezione di informazioni:

- **Flusso Server-to-Client:** Attraverso il metodo `SendCommandAsync`, il servizio espone una primitiva per l’invio di comandi strutturati (`open`, `create`). Questi vengono incapsulati in un oggetto `Message` e inoltrati al client. Per garantire l’estensibilità, il metodo generico `CallHubMethodAsync` permette l’invocazione di qualsiasi procedura esposta dall’Hub, offrendo flessibilità per evoluzioni future.
- **Flusso Client-to-Server:** Il servizio adotta un modello reattivo basato su eventi. Tramite `OnReceiveMessage`, i componenti del server MCP possono registrare handler specifici. Alla ricezione di un messaggio dal WebSocket, il servizio provvede a notificare i moduli interessati, permettendo un’elaborazione asincrona e disaccoppiata.

Architettura Semplificata e Instradamento

A differenza di architetture multi-livello complesse, il sistema attuale adotta un modello di instradamento diretto. L’`SignalrHub` funge da ripetitore trasparente: i messaggi in ingresso non subiscono processi di validazione o routing logico pesante sul server, ma vengono immediatamente inoltrati alla totalità dei client connessi tramite il metodo `Clients.All.GetMessage(message)`.

Questa scelta progettuale privilegia la bassa latenza e la semplicità di implementazione per il prototipo corrente. La protezione dell’accesso è garantita a livello infrastrutturale tramite policy di CORS (Cross-Origin Resource Sharing) definite nel punto di ingresso dell’applicazione, che limitano le connessioni ai soli domini autorizzati.

Infine, il servizio utilizza il sistema di **Dependency Injection** di ASP.NET Core per accedere ai parametri di configurazione essenziali, come il `PathBase`, permettendo al bridge di adattarsi dinamicamente all’ambiente di esecuzione senza alterazioni al codice sorgente.

3.4 Infrastruttura SignalR e WebSocket

3.4.1 Architettura Multi-Layer del Bridge

La progettazione del sottosistema di comunicazione real-time si fonda su un'architettura essenziale, strutturata per garantire semplicità e reattività attraverso il principio della separazione delle responsabilità (*Separation of Concerns*). Il sistema è suddiviso in livelli logici distinti, ciascuno incaricato di una fase specifica nel ciclo di vita del messaggio.

Dal Trasporto al Core Framework

Al vertice si colloca il **Client Layer**, rappresentato dal `Configurator3D`. Questo livello interagisce con il **Transport Layer**, il quale riduce la complessità dei protocolli di rete. Il sistema predilige l'uso di WebSocket per garantire comunicazioni *full-duplex* a bassa latenza, gestendo però in modo trasparente i fallback verso protocolli meno performanti per assicurare la compatibilità con ogni ambiente di rete.

Il coordinamento tra i protocolli e l'applicazione è affidato al **SignalR Core Layer**. In questo strato operano il *Connection Manager*, responsabile del monitoraggio delle sessioni attive, e il *Protocol Handler*, che si occupa della serializzazione dei payload in formato JSON. L'integrazione con l'SDK ufficiale permette di delegare a questo livello la gestione della robustezza e della resilienza delle connessioni a basso livello.

Logica Applicativa

Il cuore del sistema risiede nell'**Application Layer**, dove il middleware CORS permette la comunicazione cross-origin tra il server e i client web. Il punto di contatto è rappresentato dall'`IwineHub`, che riceve le invocazioni dai client. L'interazione è governata dall'interfaccia `IClient`: questo contratto *strongly-typed* definisce rigorosamente i metodi invocabili, eliminando errori di runtime e garantendo la totale *type-safety* nelle comunicazioni bidirezionali.

Infrastruttura e Persistenza Dati

A supporto dell’intera architettura agisce l’**Infrastructure Layer**. Questo livello fornisce servizi trasversali mediante il pattern della *Dependency Injection*, con metodi di estensione predisposti per future implementazioni di autenticazione e autorizzazione. Il **Data Layer** definisce il *Data Transfer Object* (DTO) per il trasporto delle informazioni, delegando al framework SignalR la gestione dello stato delle connessioni

3.4.2 Modello delle Classi e Pattern Hub

La progettazione del sottosistema di comunicazione in tempo reale si fonda sul pattern dello *Strongly-Typed Hub*, una scelta architettonica che permette di definire in modo rigoroso il contratto di interfaccia tra il server e i client connessi. In questo modello, la classe `SignalRHub` non si limita a ereditare le funzionalità base del framework ASP.NET Core, ma estende la classe astratta `Hub<T>` tipizzandola con l’interfaccia `IClient`. Questo approccio garantisce la coerenza delle chiamate a tempo di compilazione, eliminando le fragilità tipiche delle invocazioni basate su stringhe e assicurando che ogni messaggio inviato dal server rispetti una firma predefinita.

3.4. INFRASTRUTTURA SIGNALR E WEBSOCKET

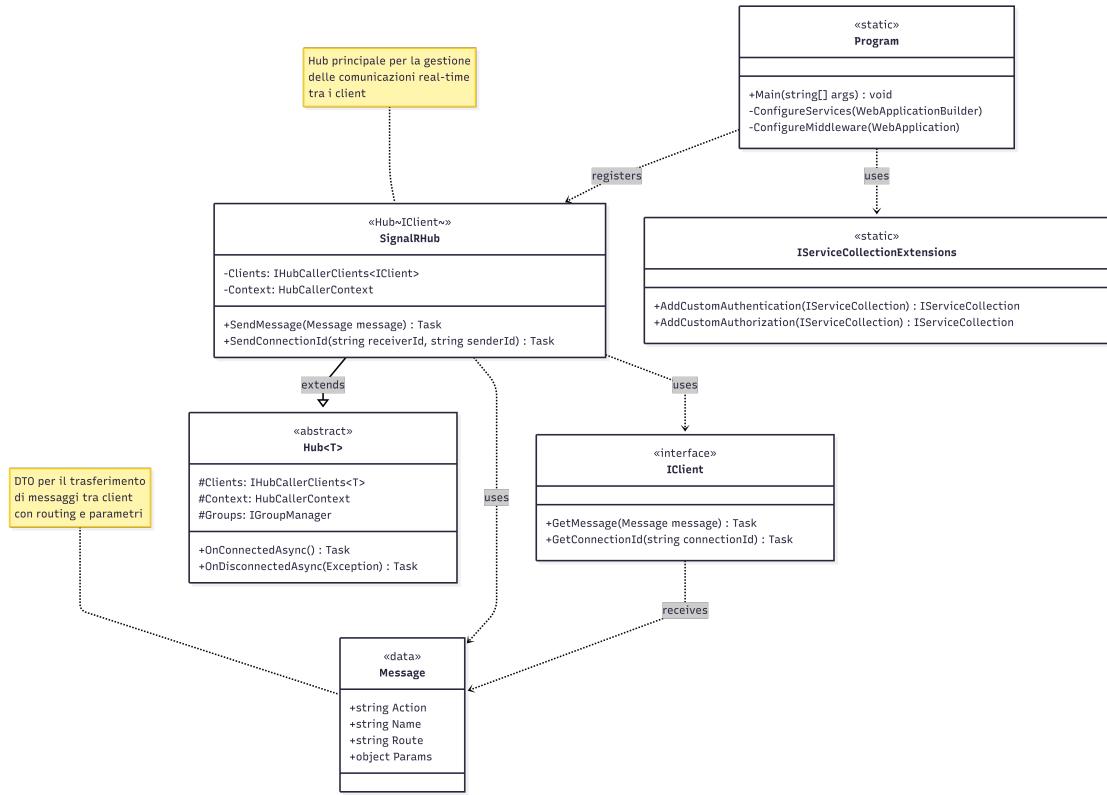


Figura 3.4: Diagramma delle classi del livello di comunicazione: struttura del SignalRHub, definizione del contratto IClient e composizione del DTO Message.

Contratti di Interfaccia e Modello Dati

L'adozione dell'interfaccia **IClient** rappresenta il punto cardine per la standardizzazione delle operazioni *server-to-client*. Essa definisce formalmente due metodi che il server può invocare sui client connessi: **GetMessage**, dedicato alla ricezione dei payload operativi, e **GetConnectionId**, utilizzato per la sincronizzazione degli identificativi di sessione in comunicazioni punto-a-punto. Questa astrazione permette di separare nettamente la logica di invio dalla tecnologia di trasporto, rendendo il sistema manutenibile e trasparente.

Il veicolo informativo tra le parti è rappresentato dal DTO (*Data Transfer Object*) denominato **Message**. Questa classe è stata progettata come un contenitore semantico flessibile, capace di trasportare l'intero contesto dell'interazione in

un'unica struttura dati. Le proprietà `Action` e `Name` permettono di determinare l'operazione richiesta (ad esempio la creazione o l'apertura di un elemento), mentre la proprietà `Route` fornisce un riferimento logico sul percorso del messaggio. La proprietà `Params` di tipo `object` consente il trasporto di metadati dinamici, assicurando che il client ricevente disponga di tutte le informazioni necessarie per elaborare correttamente il comando nel proprio contesto applicativo.

Orchestrazione dei Messaggi e Gestione della Sessione

All'interno dell'architettura, la classe `SignalRHub` assume il ruolo di orchestraatore centrale. La sua responsabilità primaria consiste nella gestione dei flussi di messaggi in ingresso e nella loro successiva distribuzione. Attraverso il metodo `SendMessage`, l'Hub riceve i pacchetti dati e, sfruttando le primitive del framework, ne esegue il broadcasting verso la totalità dei client attivi tramite il metodo `Clients.All.GetMessage`. Questa logica di propagazione circolare permette di mantenere sincronizzate le diverse istanze del configuratore senza introdurre complessi motori di instradamento intermedi, privilegiando la reattività e la semplicità di flusso.

Oltre allo smistamento dei messaggi, l'Hub facilita lo scambio degli identificativi di connessione tramite il metodo `SendConnectionId`. Questo passaggio è fondamentale per consentire ai client di conoscersi reciprocamente all'interno della rete di SignalR, ponendo le basi per interazioni mirate. Per quanto riguarda il ciclo di vita delle connessioni, il sistema si affida alle implementazioni standard fornite dalla classe base `Hub`, che gestisce in modo trasparente l'attivazione e la disattivazione delle sessioni, garantendo stabilità operativa senza la necessità di logiche di monitoraggio personalizzate.

Infrastruttura di Bootstrap e Configurazione

L'integrazione dell'`SignalRHub` nell'ecosistema dell'applicazione avviene attraverso una configurazione minimale gestita nella classe `Program`. Seguendo i principi della *Dependency Injection*, i servizi necessari al funzionamento del protocollo vengono registrati all'avvio mediante `AddSignalR()`, permettendo una gestione pulita e centralizzata delle dipendenze.

In questa fase di bootstrap, viene definito l'endpoint dedicato all'Hub tramite `MapHub<SignalRHub>("/hub")` e vengono impostate le policy di *Cross-Origin Resource Sharing* (CORS) che permettono connessioni da qualsiasi origine con credenziali, essenziali per la comunicazione tra il server e i client web distribuiti. Il sistema prevede inoltre il supporto opzionale per l'esecuzione come servizio Windows attraverso `AddWindowsService()`.

La modularità è supportata dalla classe di estensione `IServiceCollection-Extensions`, che predispone metodi stub (`AddCustomAuthentication` e `AddCustomAuthorization`) per future evoluzioni, garantendo che il sistema sia pronto a ospitare strati aggiuntivi di controllo e autorizzazione qualora i requisiti di sicurezza dovessero evolversi verso modelli più complessi.

3.4.3 Gestione del Ciclo di Vita delle Connessioni

La robustezza di un sistema *real-time* dipende in larga misura dalla capacità di gestire in modo deterministico il ciclo di vita delle connessioni client-server. L'architettura progettata si affida ai meccanismi nativi del framework SignalR, che implementa internamente un modello a stati finiti per governare ogni fase dell'interazione, dalla negoziazione iniziale fino alla disconnessione, garantendo la coerenza del canale di comunicazione anche in presenza di instabilità di rete.

Protocollo di Negoziazione e Handshake

Il processo di connessione segue una sequenza rigorosa, gestita automaticamente dal framework SignalR. La procedura ha inizio con la fase di **Negoziazione** (*HTTP Negotiate*), durante la quale il client SignalR contatta l'endpoint configurato (`/hub`) tramite una richiesta HTTP POST standard. In questa fase preliminare, il server non instaura ancora il canale persistente, ma restituisce un *Connection Token* crittografico e un elenco dei trasporti disponibili (WebSocket, Server-Sent Events, Long Polling), permettendo al client di selezionare la strategia di comunicazione più efficiente supportata dall'ambiente.

Successivamente, se le condizioni lo permettono, avviene l'**Upgrade del Transporto**. Il client richiede l'elevazione della connessione tramite l'header `Upgrade: websocket`, in conformità alla RFC 6455. Il server risponde con lo status 101

3.4. INFRASTRUTTURA SIGNALR E WEBSOCKET

Switching Protocols, trasformando la connessione HTTP effimera in un tunnel TCP persistente e bidirezionale. A questo punto, il framework assegna automaticamente un identificativo univoco alla sessione (*ConnectionId*) che può essere trasmesso al client per permettere operazioni di comunicazione mirata. Questo scambio completa l'handshake, portando il sistema in uno stato pienamente operativo.

3.4. INFRASTRUTTURA SIGNALR E WEBSOCKET

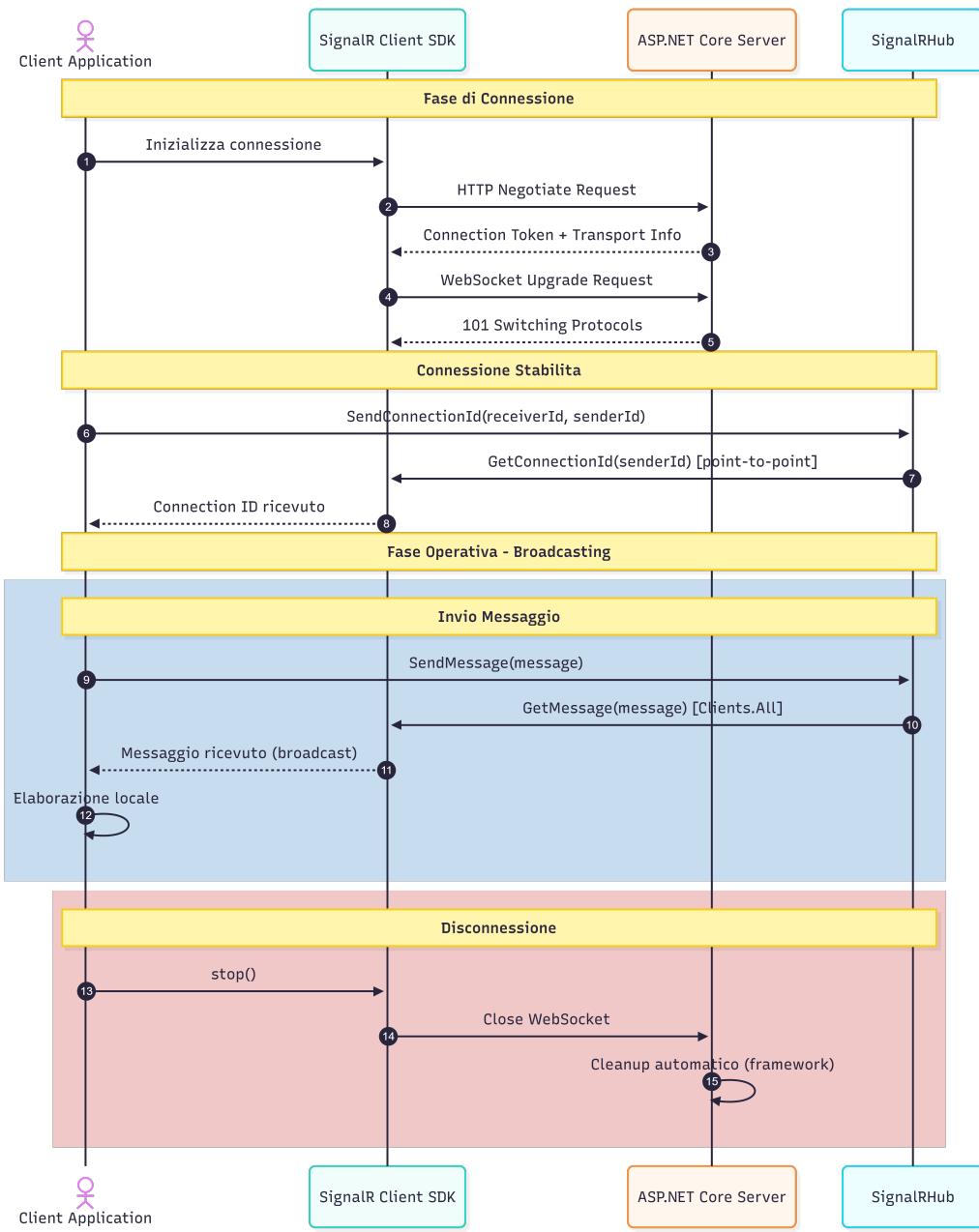


Figura 3.5: Diagramma di sequenza del flusso di connessione e messaggistica tra Configurator3D, IwineHub e framework SignalR.

Macchina a Stati della Connessione

Per modellare il comportamento dinamico del client, il framework SignalR implementa internamente una macchina a stati (*State Machine*) che copre l'intero ciclo di vita della sessione. Lo stato iniziale, **Disconnected**, rappresenta l'assenza di attività. L'invocazione del metodo di avvio innesca la transizione verso lo stato **Connecting**, durante il quale il sistema tenta di completare l'handshake entro un timeout configurato dal framework.

Il successo della negoziazione porta allo stato **Connected**, che rappresenta l'instaurazione effettiva del canale. In questa fase operativa, il client può inviare e ricevere messaggi attraverso le primitive fornite dall'Hub. Il framework gestisce automaticamente meccanismi di *Keep-Alive* (ping/pong) per monitorare la salute del canale in assenza di traffico dati, garantendo la rilevazione tempestiva di connessioni inattive.

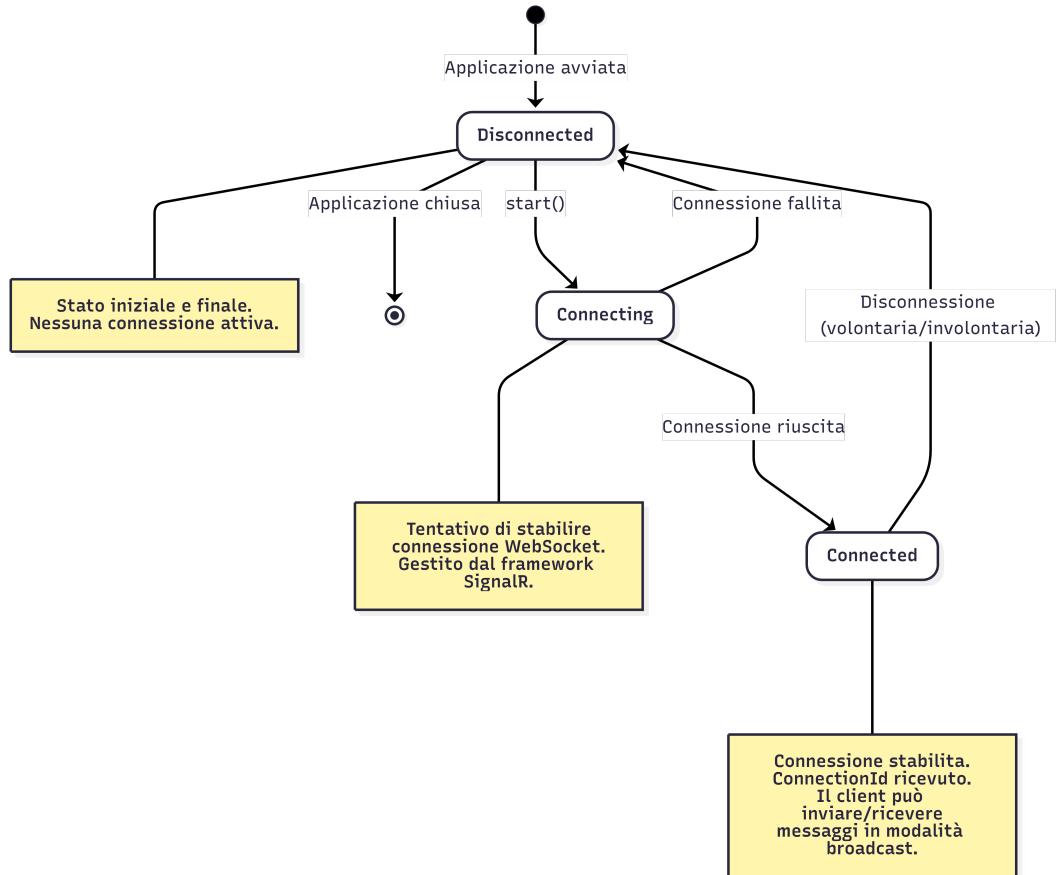


Figura 3.6: Macchina a stati della connessione SignalR: fasi di negoziazione, operatività e disconnessione gestite dal framework.

L’architettura attuale si affida alle politiche di resilienza predefinite del framework. In caso di interruzione imprevista del segnale, il sistema rileva automaticamente la perdita di connessione. Il comportamento in scenari di disservizio dipende dalla configurazione del client: nella configurazione base, il sistema notifica l’errore all’applicazione ospite che può decidere se tentare una riconnessione manuale.

La fase di **Disconnessione** può essere innescata esplicitamente dal client tramite l’invocazione del metodo di chiusura, o implicitamente dal server per timeout o condizioni anomale. In entrambi i casi, il framework esegue una procedura di pulizia, rilasciando le risorse allocate alla sessione e invalidando il *ConnectionId*,

garantendo la corretta gestione dello stato globale del sistema. Gli eventi del ciclo di vita (`OnConnectedAsync` e `OnDisconnectedAsync`) sono disponibili nella classe base `Hub<T>` per estensioni future che richiedano logiche personalizzate durante l'instaurazione o la chiusura delle connessioni.

Configurazione e Policy di Comunicazione

La configurazione del sistema avviene attraverso la registrazione dei servizi SignalR nella fase di *bootstrap* dell'applicazione. L'endpoint dell'Hub viene mappato su una rotta specifica, mentre le policy CORS vengono impostate per permettere la comunicazione tra il server e i client distribuiti su origini diverse. Questa configurazione garantisce la compatibilità con scenari di deployment in cui il frontend e il backend risiedono su domini distinti, mantenendo la sicurezza attraverso la gestione delle credenziali e degli header di autenticazione.

Il sistema è progettato per supportare estensioni future, infatti la modularità della configurazione permette di aggiungere parametri personalizzati per timeout, dimensioni dei buffer, e politiche di riconnessione avanzate senza modificare l'architettura di base. Questa flessibilità assicura che il sistema possa evolvere per soddisfare requisiti operativi più complessi mantenendo la semplicità dell'implementazione iniziale.

3.4.4 Pattern di Comunicazione: Broadcasting e Point-to-Point

L'infrastruttura di comunicazione real-time è stata progettata per offrire la massima flessibilità, supportando tecnicamente diverse topologie di instradamento dei messaggi. Tuttavia, in relazione agli obiettivi specifici di questa tesi e allo stato attuale del prototipo, è stata operata una precisa scelta implementativa riguardo il pattern di distribuzione dei dati.

Adozione del Pattern Broadcasting

Nello scenario operativo corrente, il sistema è stato configurato per utilizzare prevalentemente il pattern di Broadcasting. Tale decisione è giustificata dalla

3.4. INFRASTRUTTURA SIGNALR E WEBSOCKET

natura sperimentale del progetto, che in questa fase non prevede la gestione concorrente di sessioni multi-utente. Essendo l’interazione limitata a un singolo flusso logico tra il server MCP e l’istanza del configuratore 3D, l’invio indistinto dei messaggi a tutti i client connessi rappresenta la soluzione più efficiente e leggera, eliminando l’overhead necessario per la gestione e il tracciamento degli identificativi di sessione.

Come illustrato nel diagramma di sequenza seguente, quando il server necessita di aggiornare la scena grafica, l’Hub propaga l’evento parallelamente sull’intero canale. Il client del configuratore, essendo l’unico attore rilevante in ascolto, intercetta ed elabora il comando in autonomia, garantendo l’immediatezza dell’esecuzione senza richiedere complessi meccanismi di handshake per l’indirizzamento.

3.4. INFRASTRUTTURA SIGNALR E WEBSOCKET

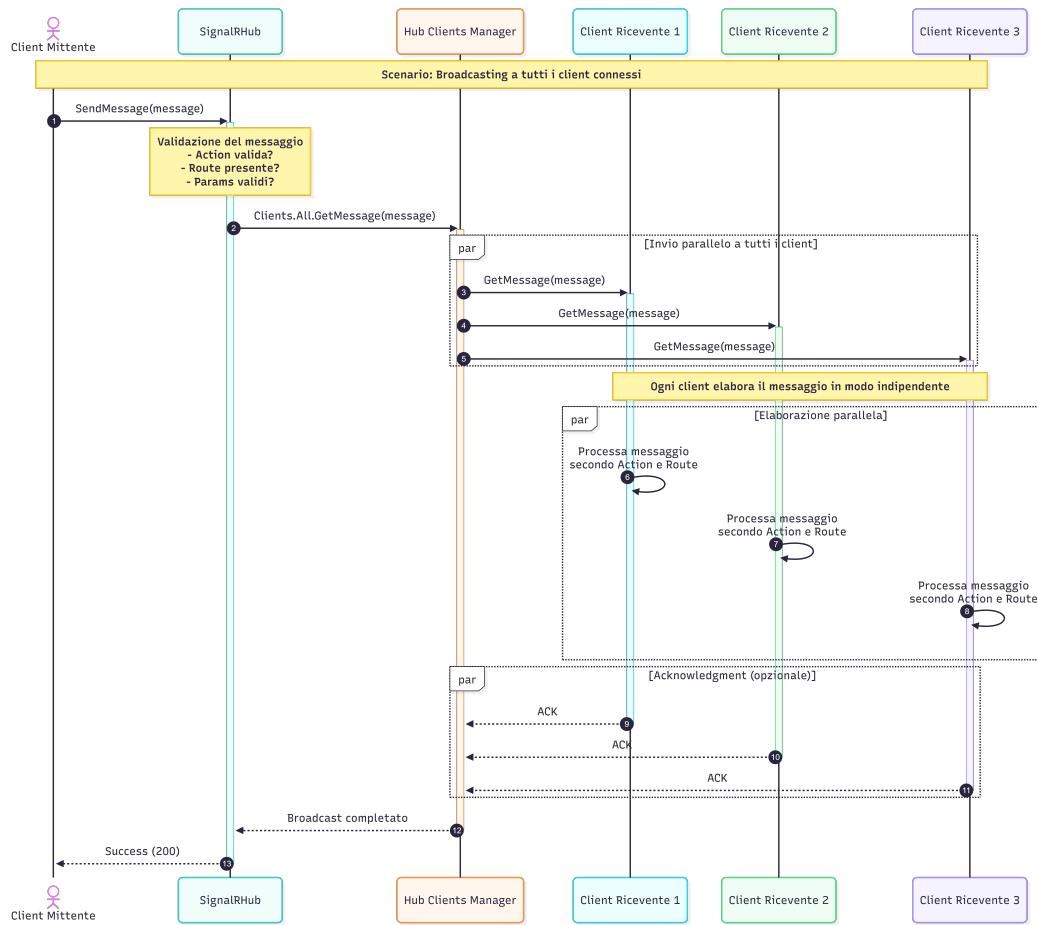


Figura 3.7: Flusso di Broadcasting: propagazione parallela del messaggio (soluzione adottata nel prototipo).

Predisposizione per la Comunicazione Point-to-Point (P2P)

Nonostante l'adozione attuale del modello diffusivo, l'architettura dell'**SignalRHub** è stata ingegnerizzata per supportare nativamente anche la comunicazione Point-to-Point (P2P). Questa capacità rappresenta un elemento cruciale per la scalabilità futura del sistema verso scenari di produzione multi-tenant.

Qualora il sistema dovesse evolvere per gestire più progettisti che operano contemporaneamente su configurazioni distinte, il Broadcasting diverrebbe inefficiente per problemi di privacy e collisione dei comandi. In tale contesto, l'architettura

è già pronta per attivare il routing diretto. Sfruttando gli identificativi univoci di sessione (*ConnectionId*) generati da SignalR, l'Hub possiede già le primitive necessarie per creare tunnel di comunicazione privati tra il server e uno specifico client. Questa predisposizione garantisce che il passaggio a un ambiente multi-utente non richiederà una riscrittura del nucleo di comunicazione, ma soltanto l'attivazione di logiche di instradamento mirato già definite nel contratto dell'interfaccia.

3.4.5 Flusso di Elaborazione Messaggi

L'efficacia dell'architettura real-time proposta risiede nella definizione di una pipeline rigorosa per il trattamento dei dati, che governa l'intero ciclo di vita del messaggio dalla sua genesi sul client fino alla ricezione e all'esecuzione sul destinatario. Questo flusso non si limita al mero trasporto di byte, ma implementa una logica strutturata di trasformazione, validazione e instradamento che garantisce l'integrità semantica delle comunicazioni all'interno del sistema distribuito.

Origine e Trasmissione: Il Lato Client

Il processo ha inizio nel livello applicativo del client mittente, dove viene costruito l'oggetto DTO (*Data Transfer Object*) **Message**. In questa fase, il sistema popola le proprietà fondamentali: l'**Action**, che definisce l'intento operativo (ad esempio "notifica" o "command"), la **Route** per l'instradamento, e l'oggetto **Params** contenente il payload specifico. Una volta invocato il metodo di invio, l'SDK client di SignalR si fa carico delle complessità di basso livello. Il framework serializza automaticamente l'oggetto nel formato concordato (tipicamente JSON) e seleziona in modo trasparente il miglior trasporto disponibile. Questa astrazione permette allo sviluppatore di ignorare i dettagli del protocollo di rete: se l'ambiente lo supporta, il payload viaggia su un canale WebSocket a bassa latenza; in caso contrario, il sistema degrada automaticamente verso Server-Sent Events o Long Polling, garantendo la consegna senza richiedere logiche condizionali nel codice sorgente.

Ricezione e Validazione Server-Side

Giunto all'Hub centrale, il messaggio subisce un processo di "ingestione". Il motore di SignalR esegue il *binding* dei dati, deserializzando il payload JSON e ricostruendo l'istanza della classe `Message` in ambiente .NET. Prima di qualsiasi elaborazione logica, il messaggio viene sottoposto a una validazione formale rigorosa. Il sistema verifica che l'oggetto non sia nullo, che l'azione sia definita e che la rotta rispetti i pattern sintattici previsti. Il fallimento di anche uno solo di questi controlli interrompe immediatamente il flusso, sollevando un'eccezione (`HubException`) che notifica al mittente l'errore di formato (Bad Request), proteggendo il server dall'elaborazione di dati inconsistenti.

3.4. INFRASTRUTTURA SIGNALR E WEBSOCKET

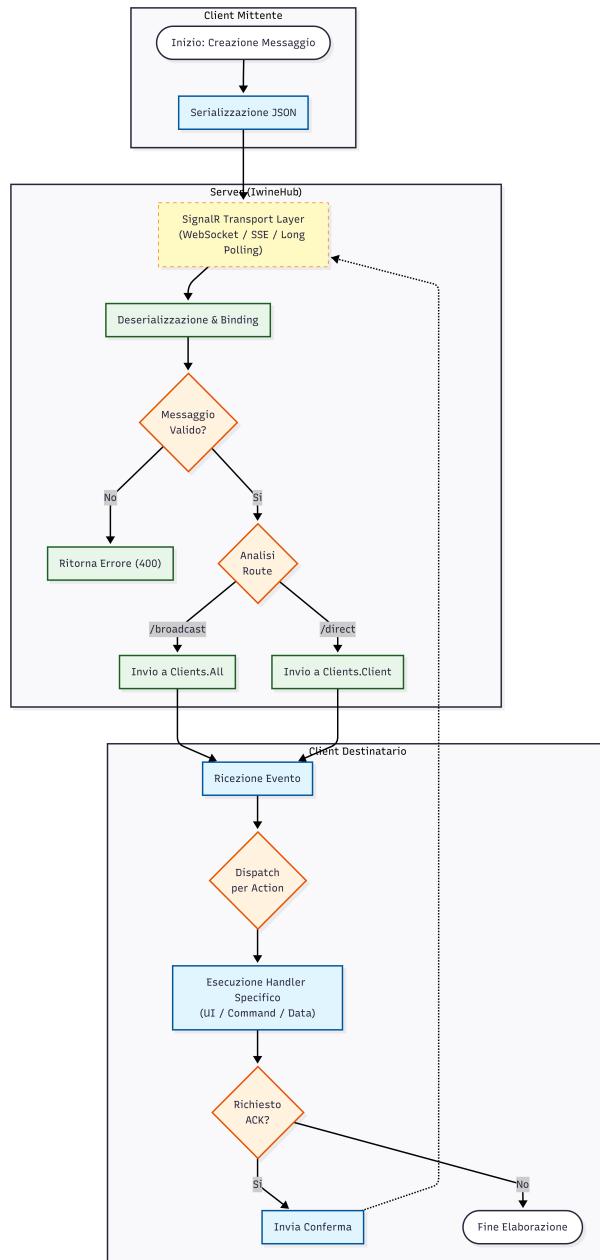


Figura 3.8: Pipeline logica di elaborazione del messaggio: dalla serializzazione all'esecuzione sul client.

Logica di Routing e Distribuzione

Superata la validazione, il componente di *Routing Engine* analizza la proprietà **Route** per determinare la strategia di destinazione. Il sistema supporta un meccanismo di *pattern matching* che distingue tre scenari principali:

1. Broadcast (/broadcast): Il messaggio è destinato all'intera platea. Il server utilizza primitive di parallelismo (`Task.WhenAll`) per inviare il dato simultaneamente a tutti i client connessi, minimizzando la latenza complessiva.
2. Direct (/direct/:id): Il sistema estrae l'identificativo del destinatario dai parametri. Viene effettuata una verifica di presenza nel contesto attivo (*Hub Context*): se il client è online, il messaggio viene instradato sul suo canale privato; in caso contrario, possono essere attivate strategie di gestione dell'assenza (come la notifica di errore al mittente).

Ricezione ed Esecuzione sul Destinatario

Il ciclo si conclude sul client ricevente. L'SDK intercetta l'evento di rete e deserializza nuovamente il contenuto in un oggetto JavaScript/TypeScript. Qui entra in gioco un meccanismo di *dispatching* basato sull'azione: uno *switch* logico analizza la proprietà **Action** e delega l'esecuzione al gestore specifico. Ad esempio, un'azione di tipo "command" potrebbe innescare l'apertura di un modello 3D, mentre una "notifica" si limiterebbe ad aggiornare l'interfaccia utente. Opzionalmente, per operazioni critiche, il protocollo può prevedere un *Acknowledgment* (ACK): terminata l'elaborazione, il client invia un segnale di ritorno al server, confermando l'avvenuto successo dell'operazione e chiudendo la transazione logica.

3.4.6 Integrazione con l'Architettura MCP

L'architettura progettata trova la sua concreta realizzazione nel flusso di integrazione che lega il server MCP al client grafico attraverso il bridge SignalR. In questo contesto, il `SignalRService` non opera più come un componente isolato, ma assume il ruolo di attuatore per gli strumenti (*Tools*) esposti dall'agente di Intelligenza Artificiale. L'obiettivo di questa integrazione è tradurre l'intento se-

mantico dell’utente, espresso in linguaggio naturale, in comandi imperativi che modificano lo stato della scena 3D in tempo reale.

Il Flusso di Esecuzione End-to-End

Il processo di interazione è stato modellato come un ciclo chiuso (*closed-loop*), essenziale per garantire all’AI il feedback necessario sul completamento delle operazioni. Tale flusso si articola in quattro fasi sequenziali:

1. Interpretazione dell’Intento (AI → MCP): Il ciclo ha inizio quando l’AI Assistant riceve un prompt testuale, ad esempio "Apri il progetto Cucina Milano". Il server MCP, analizzando la richiesta, identifica la funzione pertinente e la traduce in una chiamata di tool strutturata (es. `OpenProject`), estraendo i parametri necessari come il nome o l’ID del progetto.
2. Incapsulamento e Dispatch (MCP → SignalR): Il tool invocato non interagisce direttamente con il client, ma delega l’operazione al `SignalRService`. In questa fase avviene la traduzione dal dominio dell’AI al dominio della messaggistica: il comando viene incapsulato in un oggetto `Message` (DTO), specificando l’azione (es. "open"), il payload dati e la rotta di destinazione. Essendo il servizio registrato come Singleton, esso garantisce che tutti i tool MCP accedano alla medesima istanza del bridge, mantenendo la coerenza del canale di comunicazione.
3. Instradamento ed Esecuzione (Hub → Client): L’`SignalRHub` riceve il messaggio e, sfruttando la connessione WebSocket persistente, lo instrada verso il `Configurator3D`. Il client intercetta il comando, decodifica il payload e innesca le logiche interne di rendering, procedendo al caricamento del modello richiesto e all’aggiornamento della vista grafica.
4. Feedback Asincrono (Client → MCP): L’ultimo passaggio è cruciale per la sincronizzazione. Una volta completata l’operazione grafica, il client invia un messaggio di risposta verso il server. Il `SignalRService`, in ascolto tramite i propri handler, intercetta questo feedback e lo restituisce al tool MCP in attesa. Questo permette all’AI Assistant di confermare all’utente l’avvenuto

successo dell'operazione ("Ho aperto il progetto Cucina Milano"), chiudendo il cerchio comunicativo.

Considerazioni di Design per l'Integrazione

La stabilità di questo flusso si basa su specifiche scelte di design volte a mitigare la complessità dei sistemi distribuiti. Un aspetto critico riguarda la gestione del mapping delle connessioni: poiché l'MCP opera in un contesto prevalentemente *stateless* (ogni richiesta è indipendente), è stato necessario implementare logiche per associare correttamente le richieste dell'AI alla specifica connessione SignalR attiva del client grafico, garantendo che i comandi raggiungano l'istanza corretta. Inoltre, l'adozione di un DTO unificato (**Message**) condiviso trasversalmente tra i layer funge da contratto formale, riducendo i rischi di disallineamento nel formato dei dati. Infine, il sistema integra meccanismi di gestione dei timeout sulle operazioni asincrone: qualora il client 3D non risponda entro una finestra temporale definita (segno di un possibile blocco o disconnessione), il server MCP è programmato per interrompere l'attesa e segnalare un'eccezione all'agente AI, garantendo che il sistema non rimanga in uno stato di attesa indefinita (*deadlock*).

3.5 Sistema di Autenticazione

La sicurezza e l'integrità delle comunicazioni tra il server MCP e il backend gestionale esterno rappresentano un requisito fondamentale dell'architettura. Poiché il sistema è progettato per operare su dati sensibili, quali anagrafiche progetti e cataloghi articoli, è stato implementato un meccanismo di autenticazione robusto basato su standard di settore, capace di garantire che ogni operazione venga eseguita in un contesto autorizzato.

3.5.1 Strategia Token-Based

Il modello di sicurezza adottato si fonda sullo standard JSON Web Token (JWT), scelto per la sua natura *stateless* e per l'ampia interoperabilità in ambienti distribuiti. A differenza delle tradizionali sessioni basate su cookie, che richiedono il mantenimento dello stato lato server, l'approccio JWT delega al client, in questo

caso il server MCP, la responsabilità di conservare e presentare le credenziali di accesso sotto forma di *Bearer Token*.

Dal punto di vista della progettazione dei dati, l’interazione con il sistema di identità esterno è mediata da specifici oggetti di trasferimento dati (DTO). In particolare, il sistema non mappa l’intera struttura dell’utente remoto, ma si focalizza sull’incapsulamento della risposta di login (`LoginResponse`), estraendo e conservando esclusivamente le informazioni necessarie alla persistenza della sessione, ovvero il token di accesso e le eventuali scadenze associate.

Per garantire l’efficienza operativa e ridurre la latenza dovuta a ripetute negoziazioni delle credenziali, il design prevede una gestione della sessione di tipo *stateful* all’interno del ciclo di vita dell’applicazione. Il token, una volta ottenuto, viene persistito in un contesto statico di memoria; questa scelta architettonica permette di riutilizzare il medesimo titolo di accesso per tutte le richieste successive, evitando di dover rieseguire il login per ogni singolo comando impartito dall’agente AI.

3.5.2 Flusso di Autenticazione e Auto-Login

Il processo di autenticazione è stato disegnato per essere trasparente all’utente finale e resiliente agli errori. Il flusso operativo può essere suddiviso in fasi logiche distinte che governano l’acquisizione e l’utilizzo del token.

In fase di avvio (*bootstrap*) o alla prima richiesta operativa, il sistema recupera le credenziali di accesso. Per facilitare l’automazione e il testing, l’architettura supporta un meccanismo di Auto-Login: le credenziali predefinite sono iniettate tramite il file di configurazione (`appsettings.json`), permettendo al server MCP di autenticarsi autonomamente verso le API esterne senza richiedere un input manuale interattivo. Questa caratteristica è cruciale per scenari *server-to-server* dove non è presente un operatore umano.

Successivamente, il sistema invoca l’endpoint di autenticazione (`POST /api/login`) del backend gestionale. Alla ricezione di una risposta positiva, il componente logico estraie il JWT dal payload e lo memorizza nella variabile statica dedicata. Da questo momento in poi, ogni chiamata REST effettuata verso il sistema esterno, sia essa la lettura di un progetto o l’inserimento di un articolo, verrà automaticamente arricchita con l’header HTTP `Authorization`, contenente il *Bearer Token* valido.

Infine, la progettazione ha tenuto conto delle considerazioni di sicurezza relative alla scadenza della sessione. Sebbene il token venga mantenuto in memoria, il sistema deve essere pronto a gestire risposte di errore di tipo 401 `Unauthorized`. In tale eventualità, l'architettura prevede logicamente la necessità di invalidare il token scaduto e innescare una nuova procedura di login, garantendo la continuità del servizio senza interrompere il flusso di lavoro dell'agente intelligente.

3.6 Gestione Progetti

La gestione dei progetti rappresenta il cuore del sistema di configurazione. L'architettura deve conciliare due esigenze diverse: da un lato la necessità di accedere ai dati anagrafici memorizzati nel sistema gestionale esterno, dall'altro il bisogno di interagire dinamicamente con l'ambiente grafico 3D. Per risolvere questa sfida, è stata adottata una strategia che separa nettamente il modo in cui i dati vengono letti da come vengono utilizzati.

Modellazione Dati e Accesso tramite Repository

Il server MCP non possiede un database proprietario per salvare i progetti, ma si comporta come un "proxy" che interroga l'infrastruttura esistente. Per non legare troppo la logica dell'Intelligenza Artificiale ai dettagli tecnici delle API esterne, è stato applicato il Repository Pattern. Questo livello di astrazione permette di nascondere la complessità delle chiamate di rete. In pratica, il codice che gestisce l'AI non deve preoccuparsi di come recuperare i dati, ma interagisce con un modello interno semplificato (`Project`). Questo modello contiene solo le informazioni essenziali per l'agente, come il codice identificativo, la descrizione e la tipologia, scartando i dettagli superflui presenti nel gestionale.

I dati ricevuti dal sistema esterno vengono uniformati tramite specifici oggetti di trasporto (DTO), come `ProjectsResponse`. Questo approccio garantisce stabilità: anche se le API esterne dovessero cambiare, la struttura dati interna rimarrebbe coerente. Inoltre, centralizzare l'accesso ai dati facilita l'implementazione di logiche di ricerca flessibili, come la possibilità di cercare un progetto per nome ignorando la differenza tra maiuscole e minuscole.

L'Approccio Dual-Channel

Una caratteristica distintiva del progetto è l'utilizzo di due canali di comunicazione separati per gestire operazioni di natura diversa. Questo approccio, che possiamo definire Dual-Channel, si struttura così:

1. Canale di Lettura (REST): Viene utilizzato per le operazioni informative, come ottenere la lista dei progetti disponibili tramite `GetProjects`. Questo garantisce che l'AI abbia sempre una visione aggiornata dei dati presenti nel gestionale.
2. Canale di Azione (SignalR): Viene utilizzato per le operazioni che modificano la scena 3D, come l'apertura (`OpenProject`) o la creazione (`CreateProject`) di un progetto.

Questa distinzione è necessaria perché "aprire un progetto" non significa solo leggere un dato, ma è un comando complesso che obbliga il client grafico a scaricare i modelli 3D e preparare la scena. Il diagramma di sequenza seguente illustra bene questa cooperazione: quando l'utente chiede di aprire un progetto, il sistema usa prima il canale REST per trovare l'ID corretto del progetto, e subito dopo usa il canale SignalR per inviare al client il comando di caricamento visivo.

3.7 Gestione Articoli e Varianti

Il dominio della configurazione di prodotto non si esaurisce nella semplice selezione di un oggetto da un catalogo, ma richiede un sistema sofisticato per la gestione delle varianti e delle regole di compatibilità. La progettazione del modulo di gestione articoli è stata quindi guidata dalla necessità di astrarre la complessità del sistema gestionale sottostante, offrendo all'agente AI un'interfaccia semplificata per la manipolazione di entità complesse.

Modellazione delle Varianti e Astrazione del Catalogo

Al centro del design dei dati si colloca la definizione di modelli capaci di rappresentare la variabilità del prodotto. Poiché un articolo può assumere configurazioni

3.7. GESTIONE ARTICOLI E VARIANTI

differenti in base a finiture, dimensioni o accessori, il sistema adotta il modello `RuleSetItem` per mappare puntualmente queste specifiche. Ogni istanza di questa classe associa un codice parametro (ad esempio, il tipo di struttura) al valore dell'opzione selezionata, includendo metadati di controllo quali `IsDef`, per identificare le configurazioni predefinite, e `IsLock`, per gestire vincoli di immodificabilità imposti dalle logiche di business.

Parallelamente, la classe `VariantOptions` funge da descrittore per l'esplorazione delle possibilità offerte dal catalogo. Essa incapsula le mappe chiave-valore delle opzioni disponibili per un dato modello, permettendo al sistema di interrogare il gestionale e restituire all'utente (o all'agente intelligente) l'elenco delle personalizzazioni ammissibili. Questa struttura disaccoppia la rappresentazione interna dei dati dalla logica di persistenza, garantendo che le operazioni di ricerca (`SearchCatalogItems`) e di recupero metadati (`GetArticleInfo`) operino su DTO (*Data Transfer Objects*) standardizzati e indipendenti dalle specificità del database fisico.

3.7. GESTIONE ARTICOLI E VARIANTI

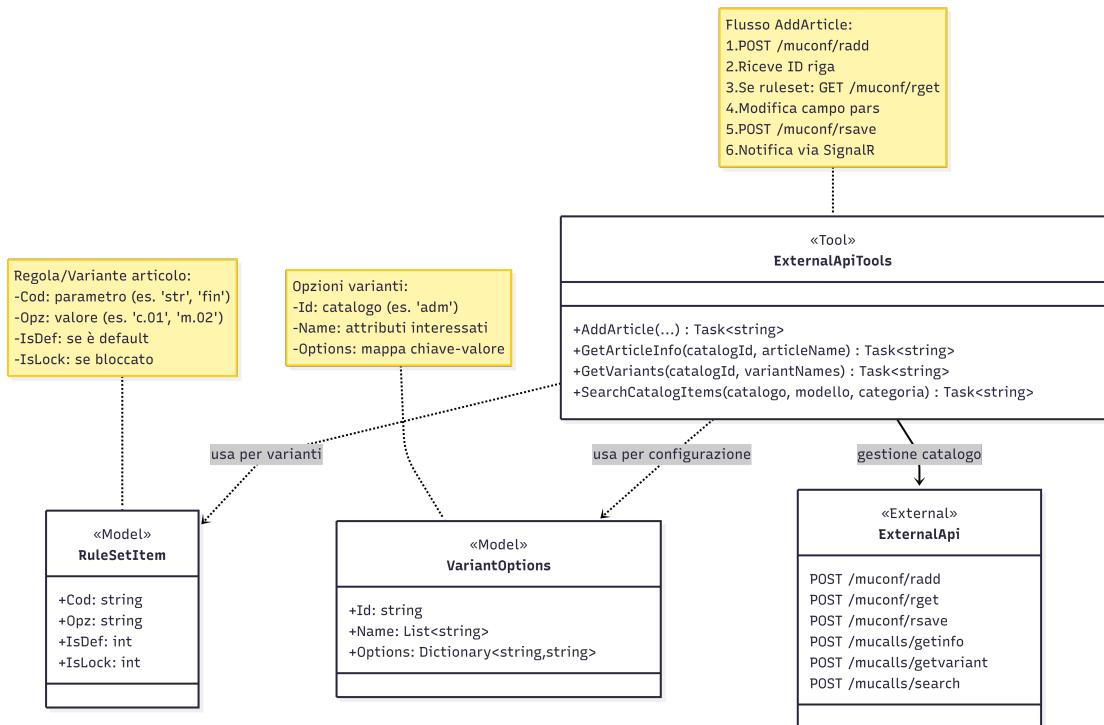


Figura 3.9: Diagramma delle classi per la gestione articoli: interazione tra i tool MCP e le primitive del gestionale esterno.

Il Pattern Transazionale di Configurazione

L’aspetto più critico nella progettazione di questo modulo risiede nel flusso operativo per l’inserimento di un nuovo articolo configurato (**AddArticle**). A differenza di una classica operazione CRUD atomica, la natura del configuratore richiede un approccio sequenziale che garantisca la consistenza dello stato tra il server MCP e il motore di regole del gestionale. Per rispondere a questa esigenza, è stato adottato il pattern architettonicale GET-MODIFY-SAVE.

Questo schema operativo, illustrato logicamente nel Diagramma 6, scomponete l’inserimento di un prodotto in una transazione a più fasi, necessaria per applicare correttamente il *RuleSet*:

1. Istanziamento (Insertion): Il processo inizia con la richiesta di inserimento

dell'articolo base tramite la primitiva `radd`. In questa fase, il sistema esterno alloca la riga d'ordine e restituisce un identificativo univoco, ma l'articolo si trova ancora nel suo stato predefinito (default).

2. Acquisizione dello Stato (Retrieval): Immediatamente dopo, il sistema effettua una lettura dello stato corrente dell'oggetto appena creato tramite `rget`. Questo passaggio è fondamentale per ottenere la stringa di configurazione (identificata come campo *pars*) che contiene l'attuale assetto delle varianti calcolato dal motore di regole.
3. Manipolazione Locale (Modification): Il server MCP interviene sulla configurazione recuperata, applicando le modifiche richieste dal prompt dell'utente (il *RuleSet*). Questa operazione avviene in memoria, modificando puntualmente i valori dei parametri senza ancora interagire con il database.
4. Persistenza e Sincronizzazione (Commit): Infine, lo stato modificato viene inviato al sistema tramite `rsave` per la persistenza definitiva. Solo al termine di questo ciclo di scrittura, il sistema notifica l'avvenuto aggiornamento al client grafico tramite SignalR, garantendo che la visualizzazione 3D rifletta esattamente la configurazione validata dal backend.

L'adozione del Command Pattern per incapsulare questa logica complessa all'interno del metodo `AddArticle` permette di esporre all'AI un unico punto di ingresso (un singolo "Tool"), mascherando l'orchestrazione delle chiamate sottostanti e garantendo l'integrità referenziale della configurazione finale.

3.8 Protocollo di Messaggistica SignalR

La robustezza dell'interazione tra il server centrale e il client grafico dipende dalla definizione di un protocollo di comunicazione rigoroso, capace di astrarre la complessità del trasporto dati e di standardizzare le interazioni. Anziché definire endpoint multipli e frammentati per ogni singola funzionalità, il design del sistema converge su un unico modello di scambio dati, il `Message`, che agisce come veicolo universale per tutte le transazioni all'interno del canale SignalR.

3.8.1 Semantica dei Messaggi e Command Pattern

Il cuore del protocollo risiede nella struttura semantica del messaggio, concepita per implementare il **Command Pattern**. In questo contesto, l'oggetto trasferito non rappresenta solamente un payload di dati, ma incapsula un'intenzione operativa specifica. La proprietà `Action` assume il ruolo di discriminante fondamentale: essa istruisce il ricevente sulla logica da attivare, distinguendo tra i diversi comandi di modifica dello stato (come `open`, `create` o `add_article`).

Questa impostazione garantisce un elevato disaccoppiamento tra le parti. Il server può inviare parametri complessi tramite l'oggetto generico `Params` e specificare il contesto dell'operazione tramite `Name` e `Route`, senza che il livello di trasporto debba conoscere i dettagli implementativi della logica di business. Tale flessibilità permette di estendere le funzionalità del sistema, aggiungendo ad esempio nuove azioni per la manipolazione della scena, senza dover alterare la firma dei metodi dell'Hub o la struttura del protocollo sottostante.

3.8.2 Orchestrazione dei Comandi Server-Side

Nel contesto dell'integrazione con l'architettura MCP, il flusso di comunicazione predominante è quello Server→Client. Il server assume il ruolo di orchestratore, traducendo gli intenti dell'agente AI in direttive esecutive per il visualizzatore grafico.

L'invio dei comandi è centralizzato nel metodo `SendHubMessage`, che agisce come unico punto di uscita verso il client connesso. In questo scenario, il modello di comunicazione è prettamente imperativo e si sviluppa nel seguente modo. Il server invia un pacchetto contenente l'azione e i parametri necessari (ad esempio, l'ID di un progetto da aprire o il codice di un articolo da inserire), e il client grafico agisce come esecutore passivo, aggiornando il rendering in risposta allo stimolo ricevuto.

Sebbene l'architettura SignalR supporti nativamente la bidirezionalità, nel design attuale questa capacità è riservata principalmente al feedback di stato (`Acknowledgment`). Il client, una volta completata l'operazione grafica richiesta (che potrebbe richiedere tempi non trascurabili per il caricamento degli asset), può notificare al server l'avvenuta esecuzione, chiudendo il cerchio logico dell'operazione

3.8. PROTOCOLLO DI MESSAGGISTICA SIGNALR

senza necessitare di complessi meccanismi di interrogazione sincrona dello stato client-side.

3.9 Flussi di Lavoro Principali

3.9.1 Caso d'Uso: Login e Recupero Progetti

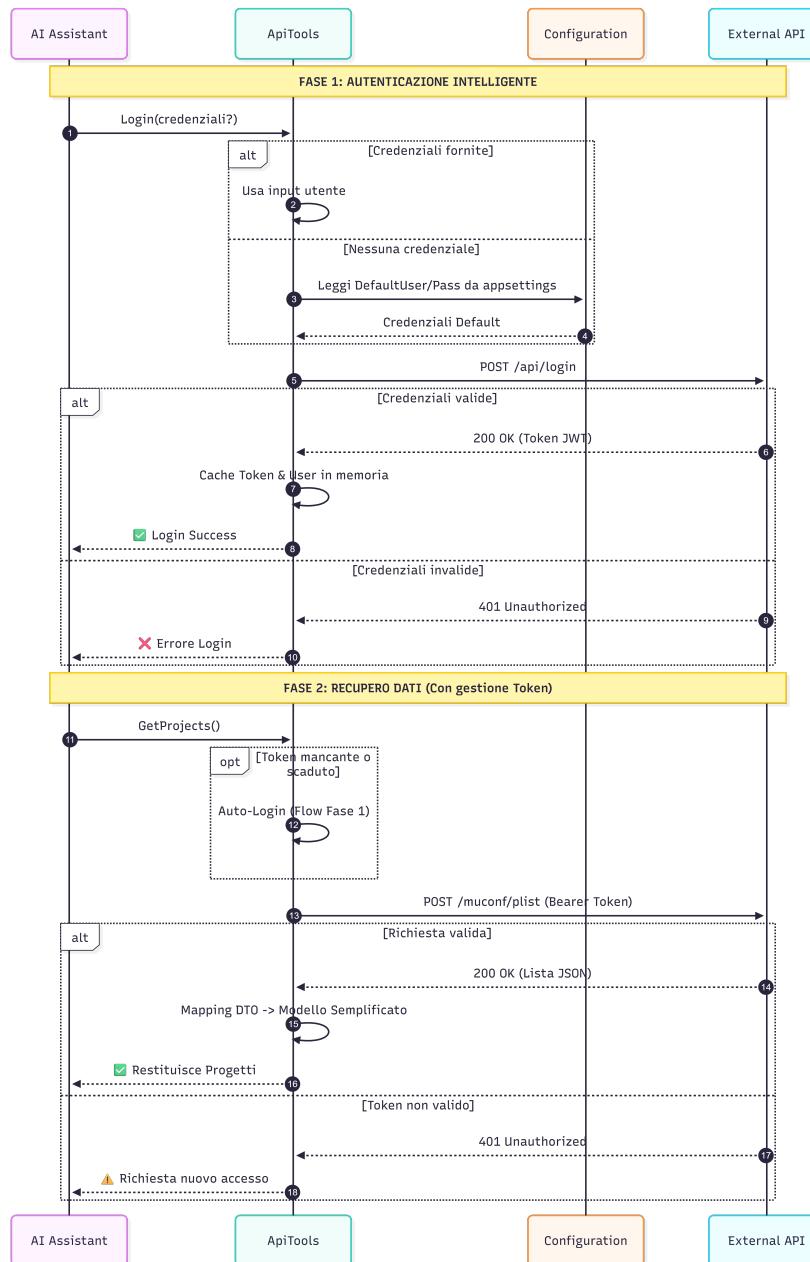


Figura 3.10: Diagramma di sequenza del flusso di autenticazione e recupero progetti: gestione intelligente del token e delle credenziali.

3.9. FLUSSI DI LAVORO PRINCIPALI

Il primo diagramma di sequenza illustra l'orchestrazione fondamentale delle interazioni tra l'agente AI e il sistema gestionale esterno, delineando due macro-fasi operative: l'autenticazione iniziale e il successivo accesso alle risorse.

La prima fase descrive il processo di login, caratterizzato da una logica di flessibilità operativa. Il sistema è progettato per gestire sia l'inserimento esplicito delle credenziali da parte dell'utente sia, in loro assenza, il recupero automatico dei parametri di accesso predefiniti dai file di configurazione. Una volta inviata la richiesta al backend remoto e ottenuta una risposta positiva, il token di sicurezza (JWT) viene estratto e persistito in una variabile statica. Questa scelta architettonica garantisce il mantenimento della sessione per le operazioni future, riducendo la necessità di autenticazioni ridondanti. Parallelamente, il successo dell'operazione viene notificato al servizio di comunicazione in tempo reale.

La seconda parte del diagramma espone il flusso di recupero dell'elenco progetti. Prima di interrogare l'API esterna, il modulo verifica preventivamente la validità del token di autenticazione, innescando una procedura di rinnovo automatico qualora questo risultasse assente o scaduto. La richiesta viene quindi inoltrata corredata dagli header di autorizzazione necessari; la risposta grezza fornita dal server viene infine deserializzata e mappata in una struttura dati semplificata, ottimizzata per essere elaborata dall'assistente intelligente. Il diagramma evidenzia inoltre la gestione degli scenari di errore, come credenziali invalide o sessioni scadute, garantendo che l'agente riceva sempre un feedback coerente sullo stato dell'operazione.

3.9.2 Caso d’Uso: Creazione e Apertura Progetto

Figura 3.11: Diagramma di sequenza del flusso di creazione e apertura progetto: orchestrazione tra API REST e canale SignalR.

Il secondo diagramma di sequenza approfondisce le dinamiche di interazione mista che caratterizzano le operazioni di modifica della scena. A differenza del semplice recupero dati, queste procedure richiedono un’orchestrazione sincrona tra il livello API (per il recupero delle informazioni) e il canale Real-Time (per l’attuazione dei comandi sul client).

3.9. FLUSSI DI LAVORO PRINCIPALI

La prima sezione illustra la creazione di un progetto: l'operazione è modellata come un comando push verso il configuratore. Il server MCP non crea direttamente il record nel database, ma istruisce il client grafico — tramite un messaggio strutturato veicolato dal bridge SignalR — affinché presenti all'utente una maschera di creazione precompilata. Questo approccio delega la responsabilità della persistenza finale al client, mantenendo il server MCP nel ruolo di "pilota" remoto.

La seconda sezione descrive l'apertura di un progetto esistente, evidenziando la necessità di una fase preliminare di risoluzione. Poiché l'utente umano tende a riferirsi ai progetti per nome (es. "Cucina Milano"), mentre il sistema richiede identificativi numerici univoci, il server MCP esegue preventivamente una query di ricerca sulle API esterne. Una volta risolto l'ID corretto, viene inviato il comando di apertura al client 3D, il quale si fa carico autonomamente del recupero degli asset pesanti (modelli 3D, texture) e del rendering della scena, confermando al termine il successo dell'operazione.

3.9. FLUSSI DI LAVORO PRINCIPALI

3.9.3 Caso d’Uso: Aggiunta Articolo con Varianti

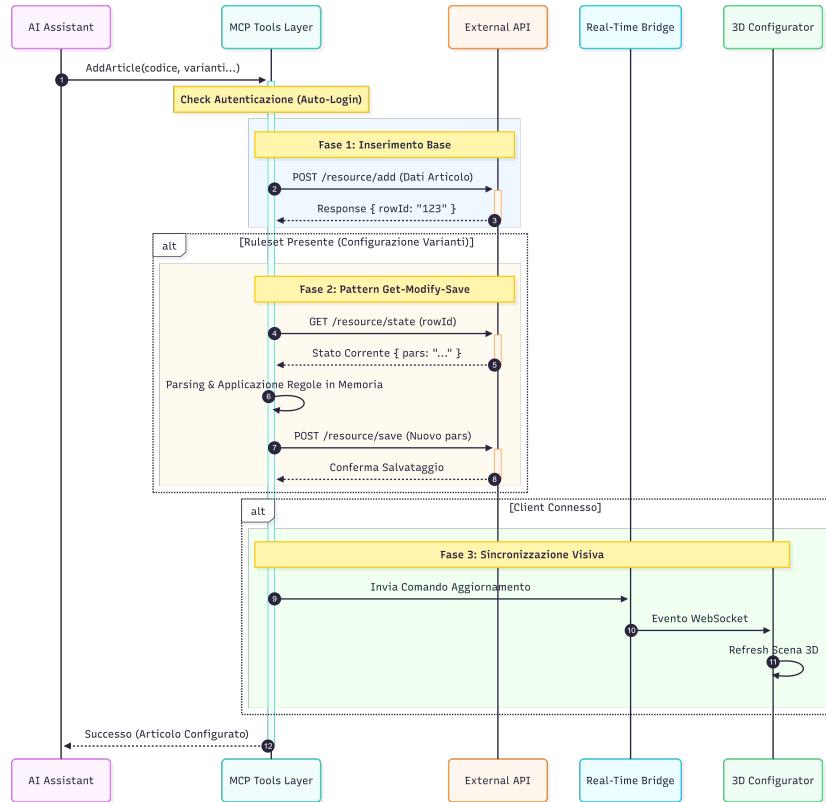


Figura 3.12: Diagramma di sequenza del flusso di aggiunta di un articolo con una variante.

Il terzo diagramma di sequenza analizza la complessità transazionale legata all’inserimento di un prodotto configurabile. A differenza di una scrittura atomica, questa operazione impone un’orchestrazione a più fasi per garantire l’applicazione corretta delle varianti (il Ruleset).

Dopo una verifica preliminare della sessione di autenticazione, il sistema procede con l’inserimento dell’articolo “neutro” tramite le API del backend gestionale. Ottenuto l’identificativo univoco della riga generata (rowId), il flusso logico valuta la presenza di regole di configurazione. Se presenti, viene attivato il pattern Get-Modify-Save: il tool recupera lo stato di configurazione predefinito dal server remoto, lo elabora in memoria applicando le varianti richieste dall’agente AI e lo persiste

nuovamente sul database. Il ciclo si conclude con la fase di sincronizzazione visiva: il server invia un comando via SignalR al client 3D, innescando l'aggiornamento della scena affinché l'utente possa visualizzare immediatamente il nuovo elemento con le finiture applicate.

3.9.4 Caso d'Uso: Comunicazione Bidirezionale

3.9.5 Caso d'Uso: Gestione Errori

Un aspetto cruciale nella progettazione di un sistema destinato all'interazione con agenti di Intelligenza Artificiale è la qualità del feedback in caso di errore. A differenza di un'interfaccia grafica tradizionale, dove l'utente umano può intuire il problema dal contesto, un agente AI necessita di messaggi di errore strutturati e "azionabili" (actionable) per poter tentare strategie di recupero autonomo. L'architettura proposta implementa una strategia di gestione delle eccezioni divisa in due livelli: la gestione delle anomalie infrastrutturali e la validazione della logica di business.

3.9. FLUSSI DI LAVORO PRINCIPALI

Gestione delle Anomalie Infrastrutturali

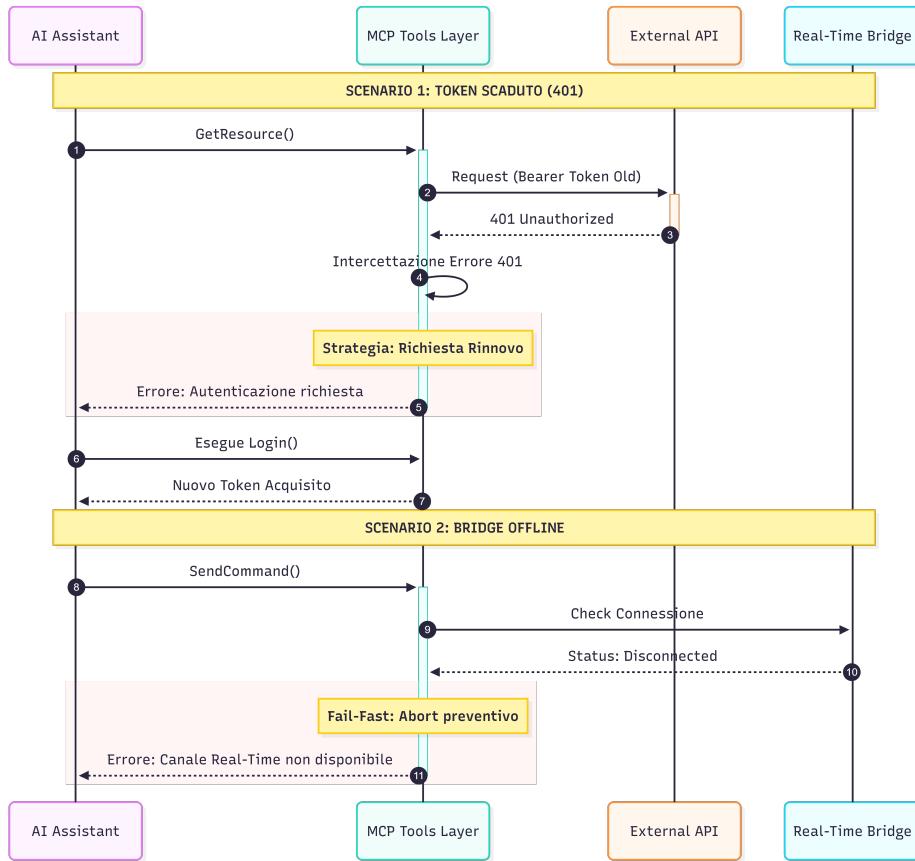


Figura 3.13: Gestione degli errori infrastrutturali: scadenza token e indisponibilità del servizio SignalR.

Il primo livello di difesa riguarda la stabilità del canale di comunicazione. Poiché il sistema dipende da token di sessione volatili e da connessioni WebSocket persistenti, è fisiologico che possano verificarsi interruzioni di servizio. Il diagramma seguente illustra due scenari critici:

1. Scadenza della Sessione: Se il token di sicurezza scade durante l'operatività, il middleware intercetta l'errore HTTP 401 e restituisce all'agente non un semplice fallimento, ma un suggerimento esplicito per effettuare il rinnovo delle credenziali.

2. Indisponibilità del Canale Real-Time: Prima di inviare comandi grafici, il sistema verifica proattivamente lo stato del bridge SignalR. Se il servizio non è raggiungibile, l'operazione viene abortita preventivamente, evitando di lasciare l'AI in attesa di un feedback che non arriverebbe mai.

Validazione Logica e Integrità dei Dati

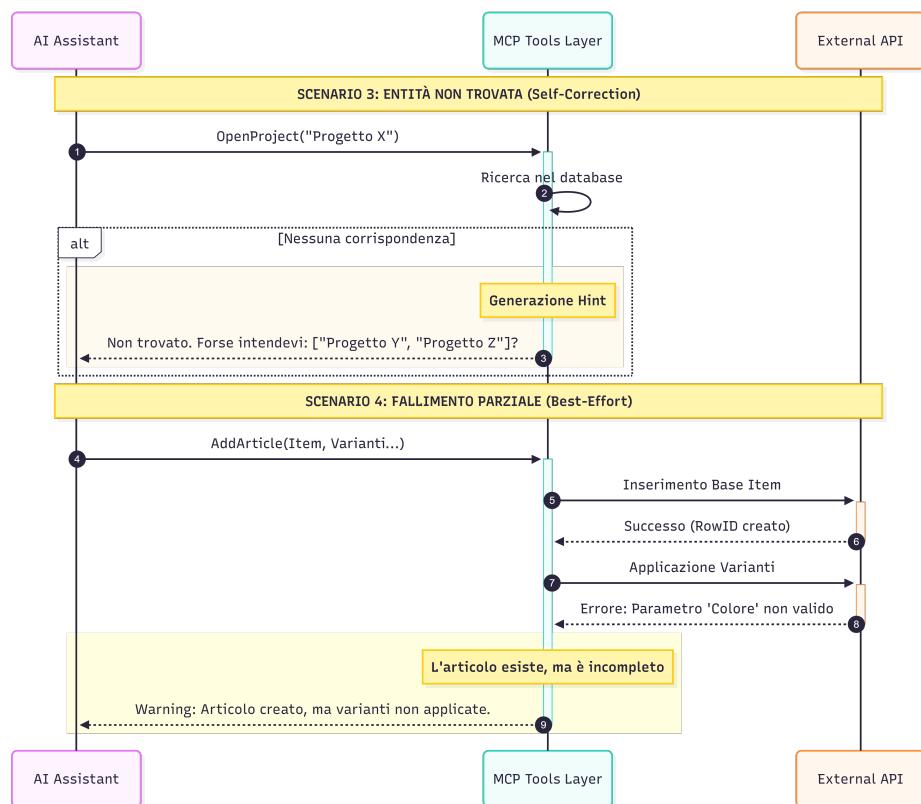


Figura 3.14: Gestione degli errori logici: entità non trovate e fallimenti parziali nella configurazione.

Il secondo livello di gestione riguarda la coerenza semantica delle richieste. Quando l'agente AI tenta di operare su entità inesistenti o fornisce parametri non conformi alle regole del configuratore, il sistema risponde con errori arricchiti di metadati contestuali.

3.9. FLUSSI DI LAVORO PRINCIPALI

Il diagramma successivo evidenzia come il sistema guidi l'agente verso la correzione dell'errore:

1. Gestione "Not Found": Nel caso di ricerca di un progetto inesistente, il sistema non si limita a restituire un codice 404, ma fornisce una lista parziale di progetti validi simili, permettendo all'LLM di correggere eventuali errori di "allucinazione" sul nome.
2. Fallimenti Parziali (Partial Success): Uno scenario complesso riguarda l'inserimento di articoli con varianti invalide. In questo caso, il sistema adotta una politica di best-effort: l'articolo base viene inserito correttamente, ma l'applicazione delle varianti errate viene segnalata come warning. Questo approccio evita il rigetto totale di operazioni lunghe e costose, informando puntualmente l'agente su quale specifico parametro ha causato l'anomalia.

Capitolo 4

Implementazione

4.1 Ambiente di Sviluppo e Setup Progetto

4.2 Implementazione MCP Server Core

4.2.1 Bootstrap e Dependency Injection

4.2.2 Implementazione Tool MCP

4.2.3 Implementazione Tool Complessi

4.3 Implementazione SignalR Service

4.3.1 Classe SignalRService e Connection Management

4.3.2 Invio e Ricezione Messaggi

4.4 Implementazione Hub (SignalR Server)

4.5 Implementazione Client Configurator3D

4.6 Gestione Configurazione e Sicurezza

Capitolo 5

Conclusioni e sviluppi futuri

Bibliografia

- [Abl20] Ably Realtime Ltd. Why websockets are better than http polling and long polling. Ably Engineering Blog, June 2020.
- [Ant24] Anthropic PBC. Introducing the Model Context Protocol. Anthropic News and Announcements, November 2024.
- [AR12] Sam Amirebrahimi and Abbas Rajabifard. An integrated web-based 3d modeling and visualization platform to support sustainable cities. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, I-4:299–304, 2012.
- [ASB25] Arash Ahmadi, Sarah Sharif, and Yaser M. Banad. Mcp bridge: A Lightweight, LLM-Agnostic RESTful Proxy for Model Context Protocol Servers. arXiv preprint arXiv:2504.08999v1, April 2025.
- [Bel22] Vaishak Belle. Logic meets learning: From aristotle to neural networks. In *Neuro-Symbolic Artificial Intelligence: The State of the Art*, pages 1–24. IOS Press, 2022.
- [BKA⁺24] Justus Bogner, Sebastian Kotstein, Daniel Abajirov, Timothy Ernst, and Manuel Merkel. RESTRuler: Towards automatically identifying violations of restful design rules in web APIs. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 1–10. IEEE/ACM, February 2024.
- [Ces24] Cesanta Software. *JSON-RPC over WS: Mongoose Examples*. Cesanta, May 2024.

- [CR24a] Branimir Cvijić and Pero Ranilović. From .net core to .net 8: A Comprehensive Analysis of Performance, Features, and Migration Pathways. In Not applicable, editor, *JITA – Journal of Information Technology and Applications*, volume 14 of *Not applicable*, pages 69–77. Pan-European University APEIRON, Banja Luka, June 2024.
- [CR24b] Branimir Cvijić and Pero Ranilović. From .net core to .net 8: A Comprehensive Analysis of Performance, Features, and Migration Pathways. *JITA – Journal of Information Technology and Applications*, 14:69–77, June 2024.
- [CSI⁺25] Gaurab Chhetri, Shriyank Somvanshi, Md Monzurul Islam, Shamyo Brotee, Mahmuda Sultana Mimi, Dipti Koirala, Biplov Pandey, and Subasish Das. Model context protocols in adaptive transport systems: A survey. *ACM Computing Surveys*, August 2025. Under Review.
- [dCdCvO22] Xavier de Carné de Carnavalet and Paul C. van Oorschot. A survey and analysis of tls interception mechanisms and motivations. *arXiv preprint arXiv:2010.16388v2*, Dec 2022. ACM Computing Surveys.
- [DZY22] Eman Daraghmi, Cheng-Pu Zhang, and Shyan-Ming Yuan. Enhancing saga pattern for distributed transactions within a microservices architecture. *Applied Sciences*, 12(12), 2022.
- [EAAS17] Mahgoub Elradi, Rodziah Atan, Rusli Abdullah, and Mohd. Hasan Selamat. A 3D e-Commerce Applications Development Model: A Systematic Literature Review. *Journal of Telecommunication, Electronic and Computer Engineering*, 9:27–33, 2017.
- [ESGK25] Abul Ehtesham, Aditi Singh, Gaurav Kumar Gupta, and Saket Kumar. A survey of agent interoperability protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP). *arXiv preprint arXiv:2505.02279v2*, May 2025. Preprint.

BIBLIOGRAFIA

- [FdSB12] Flavio S. Fogliatto, Giovani J.C. da Silveira, and Denis Borenstein. The mass customization decade: An updated review of the literature. *International Journal of Production Economics*, 138:14–25, 2012.
- [FM11] Ian Fette and Alexey Melnikov. The WebSocket protocol. RFC 6455, Internet Engineering Task Force, December 2011.
- [FSB04] Juan Diego Frutos, Eduardo Ribas Santos, and Denis Borenstein. Decision Support System for Product Configuration in Mass Customization Environments. *Concurrent Engineering: Research and Applications*, 12:131–144, 2004.
- [Goo22a] Google Cloud Platform. Best practices for managing WebSocket connections. Google Cloud Documentation, 2022.
- [Goo22b] Google Cloud Platform. Best practices for managing WebSocket connections. Google Cloud Documentation, April 2022.
- [HLE09] Anders Haug, Klaes Rohde Ladeby, and Kasper Edwards. From engineer-to-order to mass customization. *Management Research News*, 32(7):633–644, 2009.
- [JBS15] Michael B. Jones, John Bradley, and Nat Sakimura. Json web token (JWT). RFC 7519, Internet Engineering Task Force, May 2015.
- [JGVJ25] Vamshi Krishna Jakkula, Subramanya Shashank Gollapudi Venkata, and Ayush Jaiswal. Microservices with ASP.NET Core and OOP design principles. International Journal of Computer Trends and Technology, 2025.
- [JSO10] JSON-RPC Working Group. JSON-RPC 2.0 Specification. Official Specification, March 2010.
- [KD25] Sevinj Karimova and Ulviya Dadashova. The model context protocol: a standardization analysis for application integration. *UNEC Journal of Computer Science and Digital Technologies*, 1:50–59, June 2025.

BIBLIOGRAFIA

- [KU18] Kristiāns Kronis and Marina Uhanova. Performance comparison of Java EE and ASP.NET Core technologies for web API development. *Applied Computer Systems*, 23:37–44, 2018.
- [LJK17] Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. Authentication system for stateless RESTful web service. *Journal of Computational Methods in Sciences and Engineering*, 17(S1):S21–S34, June 2017.
- [McM18] Patrick McManus. Bootstrapping WebSockets with HTTP/2. RFC 8441, Internet Engineering Task Force, September 2018.
- [Med22] Sai Vaibhav Medavarapu. Real-time applications with Blazor and SignalR. *Journal of Artificial Intelligence, Machine Learning and Data Science*, 1:975–979, March 2022.
- [Mic23] Microsoft Corporation. Transport fallback in ASP.NET Core SignalR. Microsoft Learn, October 2023.
- [Mic24a] Microsoft. @microsoft/signalr - npm package, 2024. ASP.NET Core SignalR Client.
- [Mic24b] Microsoft Corporation. *IntelliSense in Visual Studio Code*. Microsoft, 2024. VS Code Documentation.
- [Mic24c] Microsoft Corporation. *TypeScript Documentation*. Microsoft, 2024. Accessed: 2024.
- [Mic24d] Microsoft Developer Division. Typescript blog, 2024. Official Development Blog.
- [Mic24e] Microsoft Learn. What's new in C# 12. Microsoft Documentation, June 2024.
- [Mic25a] Microsoft Corporation. Introduction to SignalR. Microsoft Learn, 2025.
- [Mic25b] Microsoft Learn. Overview of ASP.NET Core: Framework web and key features. Microsoft Documentation, July 2025.

BIBLIOGRAFIA

- [MM25] Model Context Protocol and Microsoft. The official C# SDK for Model Context Protocol servers and clients. GitHub Repository, February 2025. Maintained in collaboration with Microsoft.
- [Mod25a] Model Context Protocol. *Architecture Overview*. Model Context Protocol, June 2025. Documentation.
- [Mod25b] Model Context Protocol. *Understanding MCP Clients*. Model Context Protocol, June 2025.
- [Mod25c] Model Context Protocol. *Understanding MCP Servers*. Model Context Protocol, June 2025.
- [Mod25d] Model Context Protocol. *Versioning*. Model Context Protocol, June 2025.
- [Mod25e] Model Context Protocol. *What is the Model Context Protocol (MCP)?* Model Context Protocol, June 2025.
- [Moz] Mozilla Developer Network. Cross-origin resource sharing (cors) - mdn web docs. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>. Accesso: Novembre 2025.
- [Moz23] Mozilla Developer Network. Using the WebSocket API. MDN Web Docs, November 2023.
- [Moz24] Mozilla Developer Network. Static typing - mdn web docs glossary, 2024. Definitions of Web-related terms.
- [OFMWY17] Ethelbert Obinna, Faraz Fatemi Moghaddam, Philipp Wieder, and Ramin Yahyapour. A json token-based authentication and access management schema for cloud SaaS applications. arXiv preprint arXiv:1710.08281, October 2017.
- [PZWG24] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

BIBLIOGRAFIA

- [Rea24] Real-Time Experts. Signalr vs. websocket: Key differences and which to use in 2024. Protocols Comparisons Guide, September 2024. Technical Article.
- [RH25] Brandon Radosevich and John T. Halloran. Mcp safety audit: LLMs with the Model Context Protocol allow major security exploits. arXiv preprint arXiv:2504.03767v2, February 2025.
- [RJT24] Pattabi Rama Rao, Shalu Jain, and Poornima Tyagi. Enhancing web application performance: ASP.NET Core MVC and azure solutions. Technical Review, 2024. Available online.
- [ROH25] Federico Raspanti, Tanir Ozcelebi, and Mike Holenderski. Grammar-constrained decoding makes large language models better logical parsers. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL-Industry)*, pages 485–493. Association for Computational Linguistics, 2025.
- [RR25] Pattharanai Rujichaikul and Ittipon Rassameeroj. Token-based authentication monitoring system. *Journal of Cyber Security and Mobility*, 14:777–798, November 2025.
- [RSJI24] M. Raghu, Ayomide Shankeshi, Onilar Joel, and Israel. Real-time data analytics with signalr and .net: Architectural Concepts. In *Modern Digital Ecosystems: Proceedings of the International Conference*, pages 1–42, June 2024.
- [SDYD⁺23] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, et al. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 42476–42491, 2023.
- [SEKK25] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. A survey of the Model Context Protocol (MCP): Standardizing Context to Enhance Large Language Models (LLMs). Preprints.org, April 2025.

BIBLIOGRAFIA

- [SH22] Anirudh S Sundar and Larry Heck. Multimodal conversational AI: A survey of datasets and approaches. In *Proceedings of the International Conference on Conversational AI*, 2022.
- [Shi18] Krishna Shingala. Json web token (JWT) based client authentication in message queuing telemetry transport (MQTT). arXiv preprint arXiv:1903.02895, August 2018.
- [SMF⁺19] Erik Sy, Moritz Moennich, Hannes Federrath, Mathias Fischer, and Tobias Mueller. Enhanced performance for the encrypted web through tls resumption across hostnames. *arXiv preprint arXiv:1902.02531v1*, Feb 2019.
- [TC324] TC39. EcmaScript® 2024 language specification, 2024. Standard ECMA-262.
- [Tou23] Stephen Toub. Performance improvements in .NET 8. Microsoft Dev Blogs, September 2023.
- [W3C] W3C. Cors - w3c wiki. <https://www.w3.org/wiki/CORS>. Accesso: Novembre 2025.
- [WCP⁺25] Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. Mcp-bench: Benchmarking Tool-Using LLM Agents with Complex Real-World Tasks via MCP Servers. arXiv preprint arXiv:2508.20453v1, August 2025.
- [TYT20] Jerin Yasmin, Yuan Tian, and Jinqiu Yang. A first look at the deprecation of restful APIs: An empirical study. arXiv preprint arXiv:2008.12808, August 2020.
- [Zha14] Linda L. Zhang. Product configuration: a review of the state-of-the-art and future research. *International Journal of Production Research*, 52(21):6381–6398, 2014.

BIBLIOGRAFIA

Acknowledgements

Optional. Max 1 page.