redis-mcp-server (Rust)

A comprehensive Model Context Protocol (MCP) server that exposes safe, typed Redis tools over STDIO using the official Rust SDK (`rmcp`) and `redis` async client.

> Supports: keys, strings, hashes, lists, sets, sorted sets, scans, expirations, streams (XADD/XREAD), and optional RedisJSON.

📦 `Cargo.toml`

```toml
[package]
name = "redis-mcp-server"
version = "0.1.0"
edition = "2021"

[dependencies]
# Async runtime & CLI
tokio = { version = "1.39", features = ["rt-multi-thread", "macros", "signal", "io-std"] }
clap = { version = "4.5", features = ["derive", "env"] }

# MCP SDK (official)
rmcp = { version = "0.6", features = ["server", "transport-io", "macros"] }

# Redis async client
redis = { version = "0.32", features = [
  "tokio-comp", "aio", "connection-manager", "json", # JSON tools if module present
  "cluster-async", "sentinel",                        # optional future extensions
  "tls-rustls"
] }

# Serde + schema for rich tool signatures
serde = { version = "1", features = ["derive"] }
serde_json = "1"
schemars = "0.8"

# Logging & error handling
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter", "fmt"] }
```

```
anyhow = "1"
thiserror = "1"
```

---

src/main.rs

```rust
use clap::Parser;
use rmcp::{ServerHandler, model::ServerInfo, tool};
use tokio::io::{stdin, stdout};
use tracing::{error, info};

mod service;

#[derive(Parser, Debug, Clone)]
#[command(name = "redis-mcp-server", version, author, about = "MCP server
exposing Redis tools over stdio")]
struct Opts {
    /// Redis URL: redis://[:password@]host:port[/db]
    #[arg(long, env = "REDIS_URL", default_value = "redis://127.0.0.1:6379/0")]
    redis_url: String,

    /// Optional client name (used with CLIENT SETNAME)
    #[arg(long, env = "REDIS_CLIENT_NAME")]
    client_name: Option<String>,

    /// Fail if connection test (PING) does not succeed at startup
    #[arg(long, env = "REDIS_REQUIRE_PING", default_value_t = false)]
    require_ping: bool,

    /// Log verbosity (error|warn|info|debug|trace)
    #[arg(long, env = "RUST_LOG", default_value = "info")]
    log: String,
}

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let opts = Opts::parse();

    // logging
    tracing_subscriber::fmt()
        .with_env_filter(opts.log.clone())
        .with_target(false)
        .compact()
        .init();
```

```
    // Build Redis connection manager
    let svc = service::RedisService::new(&opts.redis_url,
opts.client_name.clone()).await?;

    if opts.require_ping {
        if let Err(e) = svc.ping_once().await { // sanity check before serving
            error!(error = %e, "Startup ping failed");
            anyhow::bail!(e);
        }
    }

    info!(url = %opts.redis_url, "redis-mcp-server ready on STDIO");

    // Serve MCP over stdio
    let transport = (stdin(), stdout());
    let server = svc.serve(transport).await?; // completes initialize handshake

    // Wait until the host asks us to quit
    let _ = server.waiting().await?;
    Ok(())
}
```

🔧 src/service.rs

```
use std::sync::Arc;

use redis::{aio::ConnectionManager, AsyncCommands, RedisResult};
use rmcp::{tool, schemars, ServerHandler, model::{ServerInfo, Tool, ToolKind}};
use schemars::JsonSchema;
use serde::{Deserialize, Serialize};
use serde_json::json;
use tokio::sync::Mutex;

use tracing::{instrument, warn};

#[derive(Clone)]
pub struct RedisService {
    manager: Arc<ConnectionManager>,
}

impl RedisService {
    pub async fn new(url: &str, client_name: Option<String>) ->
anyhow::Result<Self> {
        // Build client & connection manager (auto-reconnect, cheap to clone)
```

```
        let client = redis::Client::open(url)?;
        let mut manager = client.get_connection_manager().await?;

        // Optionally set a client name to help observability on the server
        if let Some(name) = client_name {
            let _: () =
redis::cmd("CLIENT").arg("SETNAME").arg(name).query_async(&mut manager).await?;
        }

        Ok(Self { manager: Arc::new(manager) })
    }

    #[instrument(skip(self))]
    pub async fn ping_once(&self) -> anyhow::Result<()> {
        let mut con = (*self.manager).clone();
        let pong: String = con.ping().await?;
        if pong.to_uppercase() != "PONG" { warn!(?pong, "Unexpected PING
reply"); }
        Ok(())
    }
}

// ======= Typed inputs / outputs for tools =======
#[derive(Debug, Deserialize, JsonSchema)]
pub struct KeyArg { pub key: String }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct SetArgs {
    /// Key to set
    pub key: String,
    /// JSON-serializable value; will be stored as a string
    pub value: serde_json::Value,
    /// EX seconds (optional)
    pub ex_seconds: Option<u64>,
    /// PX milliseconds (optional)
    pub px_millis: Option<u64>,
    /// Set only if key does not exist (NX)
    pub nx: Option<bool>,
}

#[derive(Debug, Deserialize, JsonSchema)]
pub struct ScanArgs { pub pattern: Option<String>, pub count: Option<u32> }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct HashSetArgs { pub key: String, pub field: String, pub value:
serde_json::Value }
#[derive(Debug, Deserialize, JsonSchema)]
pub struct HashGetArgs { pub key: String, pub field: String }
```

```rust
#[derive(Debug, Deserialize, JsonSchema)]
pub struct ListPushArgs { pub key: String, pub values: Vec<serde_json::Value>,
pub left: Option<bool> }
#[derive(Debug, Deserialize, JsonSchema)]
pub struct ListRangeArgs { pub key: String, pub start: i64, pub stop: i64 }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct SetMembersArgs { pub key: String }
#[derive(Debug, Deserialize, JsonSchema)]
pub struct SetAddArgs { pub key: String, pub members: Vec<String> }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct ZAddMember { pub member: String, pub score: f64 }
#[derive(Debug, Deserialize, JsonSchema)]
pub struct ZAddArgs { pub key: String, pub items: Vec<ZAddMember> }
#[derive(Debug, Deserialize, JsonSchema)]
pub struct ZRangeByScoreArgs { pub key: String, pub min: f64, pub max: f64, pub
withscores: Option<bool> }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct ExpireArgs { pub key: String, pub seconds: i64 }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct IncrByArgs { pub key: String, pub by: i64 }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct XAddArgs { pub stream: String, pub fields: serde_json::Map<String,
serde_json::Value>, pub id: Option<String> }
#[derive(Debug, Deserialize, JsonSchema)]
pub struct XReadArgs { pub stream: String, pub last_id: String, pub count:
Option<usize>, pub block_ms: Option<u64> }

#[derive(Debug, Deserialize, JsonSchema)]
pub struct JsonSetArgs { pub key: String, pub path: String, pub value:
serde_json::Value }
#[derive(Debug, Deserialize, JsonSchema)]
pub struct JsonGetArgs { pub key: String, pub path: Option<String> }

// ======= Tools implementation using rmcp #[tool] macros =======
#[tool(tool_box)]
impl RedisService {
    /// PING the server
    #[tool(name = "redis_ping", description = "Ping Redis and return PONG")]
    pub async fn tool_ping(&self) -> Result<String, String> {
        let mut con = (*self.manager).clone();
        redis::AsyncCommands::ping(&mut con).await.map_err(|e| e.to_string())
    }
```

```
    /// GET value
    #[tool(name = "redis_get", description =
"Get a string value by key; returns null if absent")]
    pub async fn tool_get(&self, #[tool(param)] key: String) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let val: Option<String> = con.get(&key).await.map_err(|e|
e.to_string())?;
        Ok(match val { Some(s) => json!({"key": key, "value": s}), None => json!
({"key": key, "value": null}) })
    }

    /// SET value with optional EX/PX/NX
    #[tool(name = "redis_set", description = "Set string value with optional
expiration and NX")]
    pub async fn tool_set(&self, #[tool(aggr)] args: SetArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        // Build low-level command for full option support
        let mut c = redis::cmd("SET");
        c.arg(&args.key).arg(args.value.to_string());
        if args.nx.unwrap_or(false) { c.arg("NX"); }
        if let Some(s) = args.ex_seconds { c.arg("EX").arg(s); }
        if let Some(ms) = args.px_millis { c.arg("PX").arg(ms); }
        let _: () = c.query_async(&mut con).await.map_err(|e| e.to_string())?;
        Ok(json!({"ok": true}))
    }

    /// DEL a key
    #[tool(name = "redis_del", description = "Delete a key, returns number of
keys removed (0 or 1)")]
    pub async fn tool_del(&self, #[tool(param)] key: String) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let removed: i64 = con.del(&key).await.map_err(|e| e.to_string())?;
        Ok(json!({"removed": removed}))
    }

    /// EXPIRE a key
    #[tool(name = "redis_expire", description = "Set TTL in seconds for a key")]
    pub async fn tool_expire(&self, #[tool(aggr)] args: ExpireArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let set: bool = con.expire(&args.key, args.seconds).await.map_err(|e|
e.to_string())?;
        Ok(json!({"ok": set}))
    }
```

```
    /// TTL for a key
    #[tool(name = "redis_ttl", description = "Get remaining TTL in seconds; -1
no expiry, -2 missing")]
    pub async fn tool_ttl(&self, #[tool(param)] key: String) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let ttl: i64 = con.ttl(&key).await.map_err(|e| e.to_string())?;
        Ok(json!({"ttl": ttl}))
    }

    /// SCAN keys
    #[tool(name = "redis_scan", description = "Incrementally scan keys with
optional pattern and count")]
    pub async fn tool_scan(&self, #[tool(aggr)] args: ScanArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut cursor: u64 = 0;
        let mut keys: Vec<String> = Vec::new();
        loop {
            let mut c = redis::cmd("SCAN");
            c.arg(cursor);
            if let Some(ref p) = args.pattern { c.arg("MATCH").arg(p); }
            if let Some(ct) = args.count { c.arg("COUNT").arg(ct); }
            // reply is (new_cursor, Vec<keys>)
            let reply: (u64, Vec<String>) = c.query_async(&mut
con).await.map_err(|e| e.to_string())?;
            cursor = reply.0;
            keys.extend(reply.1);
            if cursor == 0 { break; }
        }
        Ok(json!({"keys": keys}))
    }

    // ===== Hashes =====
    #[tool(name = "redis_hset", description = "HSET field in a hash")]
    pub async fn tool_hset(&self, #[tool(aggr)] args: HashSetArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let _: () =
redis::cmd("HSET").arg(&args.key).arg(&args.field).arg(args.value.to_string()).query_async(&mut
con).await.map_err(|e| e.to_string())?;
        Ok(json!({"ok": true}))
    }

    #[tool(name = "redis_hget", description = "HGET a field from a hash")]
    pub async fn tool_hget(&self, #[tool(aggr)] args: HashGetArgs) ->
Result<serde_json::Value, String> {
```

```
        let mut con = (*self.manager).clone();
        let val: Option<String> =
redis::cmd("HGET").arg(&args.key).arg(&args.field).query_async(&mut
con).await.map_err(|e| e.to_string())?;
        Ok(json!({"key": args.key, "field": args.field, "value": val}))
    }

    #[tool(name = "redis_hgetall", description = "HGETALL returns the full hash
as a map")]
    pub async fn tool_hgetall(&self, #[tool(param)] key: String) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let map: redis::RedisResult<redis::Value> =
redis::cmd("HGETALL").arg(&key).query_async(&mut con).await;
        match map {
            Ok(redis::Value::Bulk(items)) => {
                // convert flat list [field, value, ...] to object
                let mut obj = serde_json::Map::new();
                let mut it = items.into_iter();
                while let (Some(f), Some(v)) = (it.next(), it.next()) {
                    let fs = match f { redis::Value::Data(b) =>
String::from_utf8_lossy(&b).to_string(), _ => format!("{:?}", f) };
                    let vs = match v { redis::Value::Data(b) =>
String::from_utf8_lossy(&b).to_string(), _ => format!("{:?}", v) };
                    obj.insert(fs, serde_json::Value::String(vs));
                }
                Ok(serde_json::Value::Object(obj))
            }
            Ok(_) => Ok(json!({})),
            Err(e) => Err(e.to_string())
        }
    }

    // ===== Lists =====
    #[tool(name = "redis_lpush", description = "LPUSH/RPUSH values (left if
left=true)")]
    pub async fn tool_lpush(&self, #[tool(aggr)] args: ListPushArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut c = if args.left.unwrap_or(true) { redis::cmd("LPUSH") } else {
redis::cmd("RPUSH") };
        c.arg(&args.key);
        for v in &args.values { c.arg(v.to_string()); }
        let newlen: i64 = c.query_async(&mut con).await.map_err(|e|
e.to_string())?;
        Ok(json!({"len": newlen}))
    }
```

```
    #[tool(name = "redis_lrange", description = "LRANGE key start stop")]
    pub async fn tool_lrange(&self, #[tool(aggr)] args: ListRangeArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let vals: Vec<String> =
redis::cmd("LRANGE").arg(&args.key).arg(args.start).arg(args.stop).query_async(&mut
con).await.map_err(|e| e.to_string())?;
        Ok(json!({"values": vals}))
    }

    // ===== Sets =====
    #[tool(name = "redis_sadd", description = "SADD members to a set")]
    pub async fn tool_sadd(&self, #[tool(aggr)] args: SetAddArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut c = redis::cmd("SADD"); c.arg(&args.key); for m in
&args.members { c.arg(m); }
        let added: i64 = c.query_async(&mut con).await.map_err(|e|
e.to_string())?;
        Ok(json!({"added": added}))
    }

    #[tool(name = "redis_smembers", description = "Return all members in a
set")]
    pub async fn tool_smembers(&self, #[tool(aggr)] args: SetMembersArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let members: Vec<String> =
redis::cmd("SMEMBERS").arg(&args.key).query_async(&mut con).await.map_err(|e|
e.to_string())?;
        Ok(json!({"members": members}))
    }

    // ===== Sorted Sets =====
    #[tool(name = "redis_zadd", description = "ZADD items (score,member) into
sorted set")]
    pub async fn tool_zadd(&self, #[tool(aggr)] args: ZAddArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut c = redis::cmd("ZADD"); c.arg(&args.key);
        for it in &args.items { c.arg(it.score).arg(&it.member); }
        let added: i64 = c.query_async(&mut con).await.map_err(|e|
e.to_string())?;
        Ok(json!({"added": added}))
    }

    #[tool(name = "redis_zrangebyscore", description = "ZRANGEBYSCORE min max,
optional WITHSCORES")]
```

```
    pub async fn tool_zrangebyscore(&self, #[tool(aggr)] args:
ZRangeByScoreArgs) -> Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut c = redis::cmd("ZRANGEBYSCORE");
c.arg(&args.key).arg(args.min).arg(args.max);
        if args.withscores.unwrap_or(false) { c.arg("WITHSCORES"); }
        // We convert heterogeneous replies to JSON
        let val: redis::Value = c.query_async(&mut con).await.map_err(|e|
e.to_string())?;
        match val {
            redis::Value::Bulk(list) => {
                let mut out = Vec::<serde_json::Value>::new();
                let mut it = list.into_iter();
                while let Some(member) = it.next() {
                    if args.withscores.unwrap_or(false) {
                        if let (Some(m), Some(s)) = (to_string(member),
it.next().and_then(to_string)) {
                            out.push(json!({"member": m, "score":
s.parse::<f64>().unwrap_or(0.0)}));
                        }
                    } else if let Some(m) = to_string(member) { out.push(json!
(m)); }
                }
                Ok(json!(out))
            }
            _ => Ok(json!([]))
        }
    }

    // ===== Counters =====
    #[tool(name = "redis_incrby", description = "INCRBY key by")]
    pub async fn tool_incrby(&self, #[tool(aggr)] args: IncrByArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let newv: i64 =
redis::cmd("INCRBY").arg(&args.key).arg(args.by).query_async(&mut
con).await.map_err(|e| e.to_string())?;
        Ok(json!({"value": newv}))
    }

    #[tool(name = "redis_decrby", description = "DECRBY key by")]
    pub async fn tool_decrby(&self, #[tool(aggr)] args: IncrByArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let newv: i64 =
redis::cmd("DECRBY").arg(&args.key).arg(args.by).query_async(&mut
con).await.map_err(|e| e.to_string())?;
        Ok(json!({"value": newv}))
```

```
    }

    // ===== Streams =====
    #[tool(name = "redis_xadd", description = "XADD stream id * or explicit;
fields is object")]
    pub async fn tool_xadd(&self, #[tool(aggr)] args: XAddArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut c = redis::cmd("XADD");
c.arg(&args.stream).arg(args.id.unwrap_or_else(|| "*".to_string()));
        for (k, v) in args.fields.iter() { c.arg(k).arg(v.to_string()); }
        let id: String = c.query_async(&mut con).await.map_err(|e|
e.to_string())?;
        Ok(json!({"id": id}))
    }

    #[tool(name = "redis_xread", description = "XREAD COUNT/ BLOCK ms from
stream since last_id")]
    pub async fn tool_xread(&self, #[tool(aggr)] args: XReadArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut c = redis::cmd("XREAD");
        if let Some(ct) = args.count { c.arg("COUNT").arg(ct); }
        if let Some(ms) = args.block_ms { c.arg("BLOCK").arg(ms); }
        c.arg("STREAMS").arg(&args.stream).arg(&args.last_id);
        let v: redis::Value = c.query_async(&mut con).await.map_err(|e|
e.to_string())?;
        Ok(redis_value_to_json(v))
    }

    // ===== RedisJSON (optional) =====
    #[tool(name = "redis_json_set", description = "JSON.SET key path value
(requires RedisJSON module)")]
    pub async fn tool_json_set(&self, #[tool(aggr)] args: JsonSetArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let _: () =
redis::cmd("JSON.SET").arg(&args.key).arg(&args.path).arg(args.value.to_string()).query_async(&mu
con).await.map_err(|e| e.to_string())?;
        Ok(json!({"ok": true}))
    }

    #[tool(name = "redis_json_get", description =
"JSON.GET key [path] (requires RedisJSON module)")]
    pub async fn tool_json_get(&self, #[tool(aggr)] args: JsonGetArgs) ->
Result<serde_json::Value, String> {
        let mut con = (*self.manager).clone();
        let mut c = redis::cmd("JSON.GET"); c.arg(&args.key); if let Some(p) =
```

```
args.path { c.arg(p); }
        let raw: Option<String> = c.query_async(&mut con).await.map_err(|e|
e.to_string())?;
        match raw { Some(s) =>
serde_json::from_str::<serde_json::Value>(&s).map_err(|e| e.to_string()), None
=> Ok(serde_json::Value::Null) }
    }
}

fn to_string(v: redis::Value) -> Option<String> {
    match v { redis::Value::Data(b) =>
Some(String::from_utf8_lossy(&b).to_string()), redis::Value::Bulk(mut inner) if
inner.len()==1 => match inner.remove(0) { redis::Value::Data(b) =>
Some(String::from_utf8_lossy(&b).to_string()), _ => None }, _ => None }
}

fn redis_value_to_json(v: redis::Value) -> serde_json::Value {
    match v {
        redis::Value::Nil => serde_json::Value::Null,
        redis::Value::Int(i) => json!(i),
        redis::Value::Data(b) => json!(String::from_utf8_lossy(&b).to_string()),
        redis::Value::Bulk(items) =>
serde_json::Value::Array(items.into_iter().map(redis_value_to_json).collect()),
        other => json!(format!("{:?}", other))
    }
}

#[tool(tool_box)]
impl ServerHandler for RedisService {
    fn get_info(&self) -> ServerInfo {
        ServerInfo {
            name: Some("redis-mcp-server".into()),
            version: Some(env!("CARGO_PKG_VERSION").into()),
            instructions: Some("Tools for safe Redis access (strings, hashes,
lists, sets, zsets, streams, JSON). Always pass explicit parameters; large scans
may be truncated by clients.".into()),
            ..Default::default()
        }
    }
}
```

🧪 Quick local run

```
# 1) Ensure Redis is running locally (or set REDIS_URL)
export REDIS_URL="redis://127.0.0.1:6379/0"

# 2) Build & run (stdio)
cargo run --release -- --require_ping
```

> The server speaks MCP over **STDIO**; connect it from any MCP client (Claude Desktop, VS Code's Copilot Chat MCP, Cursor, Cline, etc.).

🔌 Example MCP client configuration (Claude Desktop)

Paste into your MCP settings UI (Developer → MCP Servers) and adjust the command path and `REDIS_URL`:

```
{
  "mcpServers": {
    "redis-mcp": {
      "command": "/absolute/path/to/redis-mcp-server",
      "args": [],
      "transport": { "stdio": {} },
      "env": {
        "REDIS_URL": "redis://:password@127.0.0.1:6379/0",
        "RUST_LOG": "info"
      }
    }
  }
}
```

> VS Code also supports adding MCP servers via `code --add-mcp '{"name":"redis-mcp",...}'` or a workspace `mcp.json`.

🧰 Example tool calls (from an MCP host)

- `redis_set` → `{ key: "greeting", value: "hello", ex_seconds: 60 }`
- `redis_get` → `"greeting"`
- `redis_scan` → `{ pattern: "user:*", count: 500 }`
- `redis_hset` / `redis_hget` / `redis_hgetall`
- `redis_lpush` / `redis_lrange`
- `redis_sadd` / `redis_smembers`

- `redis_zadd` / `redis_zrangebyscore { withscores: true }`
- `redis_xadd` / `redis_xread`
- `redis_json_set` / `redis_json_get`

🪪 Security & safety notes

- The server exposes **only** specific Redis commands via typed tools, not arbitrary command pass-through.
- Use `REDIS_URL` with TLS (e.g. `rediss://`) and proper auth in production.
- For multi-tenant hosts, consider network policy around the Redis endpoint.

🧱 Containerfile (Podman-friendly)

```
# syntax=docker/dockerfile:1.7
FROM rust:1.80 AS builder
WORKDIR /app
COPY . .
RUN --mount=type=cache,target=/usr/local/cargo/registry \
    --mount=type=cache,target=/app/target \
    cargo build --release

FROM debian:bookworm-slim
RUN useradd -m app
COPY --from=builder /app/target/release/redis-mcp-server /usr/local/bin/redis-mcp-server
USER app
ENV REDIS_URL=redis://host.docker.internal:6379/0 \
    RUST_LOG=info
ENTRYPOINT ["/usr/local/bin/redis-mcp-server"]
```

Build & run with Podman

```
podman build -t redis-mcp-server .
podman run --rm -e REDIS_URL=redis://host.containers.internal:6379/0 -it redis-mcp-server
```

🖨️ Roadmap ideas (easy extensions)

- Add **cluster** and **sentinel** connection modes based on flags/env.
- Implement **pub/sub** as a long-running MCP *prompt* or streaming content.

• Expose **FT.SEARCH** and **vector** queries for Redis Stack.
• Optional **rate limits** per tool (simple token-bucket) to protect the DB. ```