

flame-mcp

Reference Guide

Natural language control for Autodesk Flame via Claude + MCP

Covers: Architecture · Flame menus · MCP tools · Embedded Chat widget
RAG documentation search · Self-improvement system · Token tracking
Common Flame patterns · Troubleshooting · Compatibility

1 · Overview

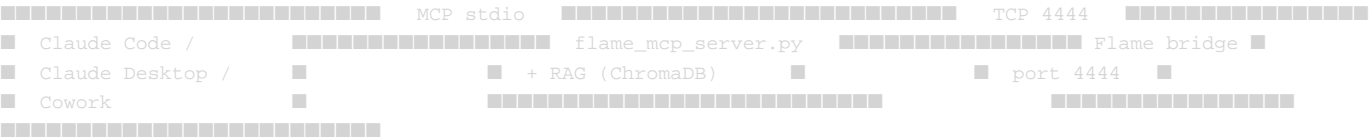
flame-mcp connects **Claude** (Code, Desktop, or Cowork) to **Autodesk Flame** via a lightweight Python bridge and the **Model Context Protocol (MCP)**. You describe what you want in plain English — Claude translates it into Flame Python API calls and executes them live, streaming results back to you.

```
You: "Delete all reels named TEST from Default Library"
      ↓
  search_flame_docs("delete reel by name") ← RAG lookup
      ↓
  execute_python(code) ← runs inside Flame
      ↓
  "Deleted: ['TEST', 'TEST2', 'DESKTOP_TEST']" ← result back to Claude
```

The system works identically in all three Claude contexts: **Claude Code** (terminal), **Claude Desktop**, and **Cowork**. The workflow rules are embedded directly in the MCP server so every client behaves the same without additional configuration.

Architecture

Component	File	Runs in	Description
TCP Bridge	hooks/flame_mcp_bridge.py	Autodesk Flame	Python hook loaded at Flame startup. Starts a TCP server on port 4444.
MCP Server	flame_mcp_server.py	macOS terminal / Claude Code	FastMCP server exposing 8 tools. Translates natural language to Flame Python API calls.
RAG Index	rag/	macOS (venv)	ChromaDB vector store built from FLAME_API.md. Searched before executing Python code.



2 · Installation

Requirements

Requirement	Version	Notes
macOS	12+	Tested on Sonoma / Sequoia
Autodesk Flame	2025+	2026 fully tested
Python	3.11+	For the MCP server venv
Node.js	v22+	Required by Claude Code
Claude Code	2.x	npm install -g @anthropic-ai/claude-code
Claude account	Pro/Max	Or Anthropic API key

Automatic installation (recommended)

```
git clone https://github.com/abrahamADSK/flame-mcp.git
cd flame-mcp
chmod +x install.sh && ./install.sh
```

The installer:

- 1 · Creates Python virtual environment and installs dependencies (mcp, chromadb, sentence-transformers)
- 2 · Copies the Flame hook to /opt/Autodesk/shared/python/ (requires sudo)
- 3 · Registers the MCP server with Claude Code
- 4 · Builds the initial RAG documentation index

Manual installation

```
# 1. Clone & venv
git clone https://github.com/abrahamADSK/flame-mcp.git && cd flame-mcp
python3 -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt --no-user

# 2. Build RAG index
python rag/build_index.py

# 3. Install Flame hook (requires sudo)
sudo cp hooks/flame_mcp_bridge.py /opt/Autodesk/shared/python/

# 4. Register MCP server with Claude Code
claude mcp add flame -- "$(pwd)/.venv/bin/python" "$(pwd)/flame_mcp_server.py"

# 5. Optional: Claude Desktop
#   Edit claude_desktop_config.json with correct paths, then:
cp claude_desktop_config.json ~/Library/Application\ Support/Claude/

# 6. Rebuild hook after any update
sudo cp hooks/flame_mcp_bridge.py /opt/Autodesk/shared/python/
# Then in Flame: MCP Bridge → Reload hook
```

Flame hook search paths (loaded in this order): `$DL_PYTHON_HOOK_PATH` · `/opt/Autodesk/shared/python/` · `/opt/Autodesk/python/` · `/opt/Autodesk/user/python/`. This project installs to `shared/python/` so it works across all Flame versions.

3 · Flame Menu — MCP Bridge

When Flame starts and the hook is loaded, an **MCP Bridge** submenu appears in Flame's main menu bar. The menu title shows the bridge status at a glance and updates every time you open it.

```
MCP Bridge  [● Active]
■■■ Status: ● Active — port 4444
■■■ Start bridge
■■■ Stop bridge
■■■ Restart bridge
■■■ Claude Chat  (embedded)      ← Qt window, no terminal needed
■■■ Launch Claude (terminal)...  ← opens Terminal.app with venv
■■■ Reload hook                  ← hot-reload, no Flame restart
■■■ Connection test              ← TCP round-trip latency check
■■■ View log...                  ← opens bridge log in TextEdit
```

Menu item	Description
Status: ● Active — port 4444	Reads the current bridge state and displays it. ● Active = listening. ■ Inactive = stopped. Updates on every me
Start bridge	Starts the TCP listener on port 4444 in a background thread. The bridge accepts one connection at a time, pro
Stop bridge	Sends a shutdown signal to the background thread and closes the socket.
Restart bridge	Equivalent to Stop + Start. Use if the bridge becomes unresponsive.
Claude Chat (embedded)	Opens the Qt chat window (see Section 4). Calls the Anthropic API directly from inside Flame — no terminal or
Launch Claude (terminal)...	Writes a launch script to ~/Library/Caches/flame-mcp/ and opens it in Terminal.app. The script activates the ve
Reload hook	Hot-reloads flame_mcp_bridge.py inside Flame using importlib. Use this after deploying a new version with suc
Connection test	Sends 'print("OK")' through the TCP bridge and shows round-trip latency. Confirms end-to-end connectivity: bri
View log...	Opens ~/Projects/flame-mcp/logs/flame_mcp_bridge.log in TextEdit. Shows every EXEC call, result snippet, an

4 · Embedded Claude Chat Widget

The **Claude Chat (embedded)** menu item opens a native Qt window that lives inside Flame. You can control Flame with natural language without needing a terminal, Claude Code, or any external application.

How it works

Step	Details
1 · API key	Reads ANTHROPIC_API_KEY from the process environment. If not found, checks ~/Projects/flame-mcp/.env (format: ANTHROPIC_API_KEY=)
2 · API call	Sends the conversation to the Anthropic API using stdlib urllib — no extra Python packages required. Model: claude-sonnet-4-5-2025-07-30
3 · Tool execution	When Claude calls a tool (execute_python, search_flame_docs, etc.), the widget connects to the local TCP bridge on port 4444 and executes the tool
4 · UI updates	A QTimer fires every 40ms and drains a thread-safe queue, appending coloured chat bubbles to the display. This avoids cross-threading
5 · Ctrl+Return	An event filter (QObject subclass, compatible with PySide2 and PySide6) catches Ctrl+Return in the input field and sends the message to the API

Qt compatibility

The widget detects whether Flame uses **PySide2** (Flame 2023–2025) or **PySide6** (Flame 2026+) and imports accordingly. The event filter class is built at runtime as a proper QObject subclass — PySide6 strictly requires this for installEventFilter.

QEvent enum paths differ between versions: PySide2 uses QEvent.KeyPress, PySide6 uses QEvent.Type.KeyPress. The filter resolves this with: getattr(QtCore.QEvent, 'Type', QtCore.QEvent).KeyPress

Singleton behaviour

Only one chat window can be open at a time. The instance is stored in _chat_instance at module level to prevent garbage collection and ensure the window stays alive after the menu action returns.

5 . MCP Tools

Claude has access to 8 tools exposed by flame_mcp_server.py via FastMCP. All responses include a token-tracking footer. session_stats() is always the last tool call in every response.

Tool	Arguments	Description
<code>execute_python</code>	<code>code: str</code>	Execute arbitrary Python code inside Flame. Full flame module access. Must always end with <code>print()</code> .
<code>get_project</code>	<code>force: bool</code>	Return name, frame rate, resolution (widthxheight), and bit depth of the active project.
<code>list_libraries</code>	<code>count: int</code>	List all libraries in the active workspace with their reel counts.
<code>list_reels</code>	<code>library: str</code>	Library name: library (if name given) or across all libraries.
<code>get_flame_version</code>	<code>code: str</code>	Return the running Flame version string (e.g. '2026.0.0.0').
<code>search_flame_docs</code>	<code>query: str</code>	Search over FLAME_API.md. Returns top 3 matching sections with relevance scores.
<code>learn_pattern</code>	<code>description: str</code> <code>code: str</code>	Add sample working code pattern to FLAME_API.md and rebuild the RAG index in the background.
<code>session_stats</code>	<code>code: str</code>	Full session summary: exec calls, RAG calls, patterns learned, tokens used/saved. Always called last.

Mandatory tool workflow (embedded in server instructions)

The FastMCP server embeds workflow rules via the `instructions=` parameter. Every Claude context (Code, Desktop, Cowork) receives these rules automatically — no per-client configuration needed.

```
Rule 1: ALWAYS call search_flame_docs BEFORE execute_python.
        Use a short query: "delete reel", "import clip", "create batch group".
        Skip only for trivially simple calls (e.g. print project name).

Rule 2: Use the correct object hierarchy:
        ws = flame.projects.current_project.current_workspace
        ws.libraries ← CORRECT
        flame.projects.current_project.libraries ← WRONG (returns None)

Rule 3: Never call flame.batch.render() directly — it crashes Flame.
        Use: flame.schedule_idle_event(render_fn)

Rule 4: Always end execute_python code with print().
        Results are only visible through stdout capture.

Rule 5: Keep code minimal — long loops can block Flame's main thread.

Rule 6: On failure, do NOT retry the same approach. Try a different method.

Rule 7: ALWAYS call session_stats as the LAST tool call of every response.

Rule 8: SELF-IMPROVEMENT — if search_flame_docs returned max score < 60%
        AND execute_python succeeded:
        → call learn_pattern(description, code) BEFORE session_stats.
```

6 · RAG Documentation Search

Before writing any `execute_python` code, Claude searches a local semantic index built from **FLAME_API.md**. This retrieves the most relevant API patterns rather than loading the entire documentation — saving ~1300 tokens per call.

Components

File	Purpose
FLAME_API.md	Source of truth. All Flame API patterns, object hierarchy, utility functions, and common workflows. Manually maintained.
rag/build_index.py	Splits FLAME_API.md into chunks by section header, embeds them using sentence-transformers, stores in ChromaDB.
rag/search.py	<code>search(query, n_results=3) → (text, max_relevance_int)</code> . Lazy singleton for collection. Logs every query + scores to log.
rag/index/	ChromaDB persistent store. Git-ignored. Rebuild after pulling new FLAME_API.md: <code>python rag/build_index.py</code>
logs/flame_rag.log	Every RAG query, top-3 results with relevance %, char/token counts. Used for debugging and coverage analysis.

Relevance score interpretation

Score range	Meaning	Action
≥ 70%	Pattern is well-documented. High confidence.	Claude uses it directly. No <code>learn_pattern</code> needed.
60–69%	Partial match. Pattern exists but may be incomplete.	Claude proceeds cautiously. Consider <code>learn_pattern</code> if code differs significantly.
< 60%	Pattern NOT documented. Low confidence.	Claude is warned in-line. After successful <code>execute_python</code> , <code>learn_pattern</code> is called automatically.
< 45%	No relevant documentation found.	Claude must reason from first principles. <code>learn_pattern</code> is critical after success.

Token savings

Loading all of FLAME_API.md costs ~1500 tokens per call. A RAG search returning 3 relevant chunks typically costs ~200 tokens. At 80% session efficiency (typical), a 10-call session saves ~10 000 tokens compared to full-doc loading.

Log example:

```
[11:50:13] Index loaded — 46 chunks
[11:50:13] QUERY: 'delete reel folder from library'
[11:50:13]   → [50%] FLAME_API.md :: Find a reel inside a library
[11:50:13]   → [49%] FLAME_API.md :: list all libraries and their reels
[11:50:13]   → [44%] FLAME_API.md :: create library + reel + import
[11:50:13]   → returned 3 chunks, ~813 chars (~203 tokens saved vs full doc)

# After learn_pattern was called, next session:
[next-session] QUERY: 'delete reel folder from library'
[next-session]   → [82%] FLAME_API.md :: Auto-learned: delete reel by name from library
[next-session]   → [78%] FLAME_API.md :: Delete / Remove Objects
```


9 · Safety — Known Crashers & Blocked Patterns

execute_python automatically scans code for patterns known to crash Flame before sending it to the bridge. If a dangerous pattern is detected, the tool returns a ■ Blocked message with the correct alternative — the code is **never executed**.

Blocked patterns

Pattern	Why it crashes	Safe alternative
len(flame.projects)	flame.projects does not have no len(). Raises TypeError	Use flame.projects.current_project.name for active project.
for p in flame.projects:	flame.projects does not iterable. Raises TypeError	Use os.listdir('/opt/Autodesk/project') to enumerate all projects.
flame.projects[0]	flame.projects is not subscriptable.	Use flame.projects.current_project directly.
flame.projects.current_project.libraries	flame.projects.current_project is None — AttributeError	Use flame.projects.current_workspace; ws.libraries
flame.projects.current_project.render	Blocks Flame's main thread. Freezes or crashes Flame.	Use flame.projects.current_project.render_event(render_fn) with Background Reactor.
import wiretap	Wiretap module is crash-prone for general scripting.	Use the standard flame module. Call search_flame_docs for patterns.
dir(flame.projects)	Spurious API discovery leads to untested code.	Call search_flame_docs(query) — returns verified patterns only.

How blocking works

```
# Claude sends this code:
for project in flame.projects:
    print(project.name)

# execute_python response (code never reaches Flame):
■ Blocked — contains pattern(s) known to crash Flame:

    • flame.projects is not iterable — iterating it crashes Flame.
      ■ Instead: Use flame.projects.current_project for the active project,
        or os.listdir('/opt/Autodesk/project') to enumerate all projects.

Revise the code and try again.
If unsure of the correct approach, call search_flame_docs first.
```

List all Flame projects — correct pattern

```
import os, flame

# List all projects from filesystem (no API, no crash)
base = "/opt/Autodesk/project"
projects = sorted(d for d in os.listdir(base)
                  if os.path.isdir(os.path.join(base, d)) and d != "project.db")

for p in projects:
    print(p)

# Currently open project
print(f"Active: {flame.projects.current_project.name}")
```

Extending the blocklist

The blocklist lives in `_DANGEROUS_PATTERNS` in `flame_mcp_server.py` — a list of (regex, reason, alternative) tuples. Add new entries whenever a crash pattern is discovered. Changes take effect immediately without restarting Claude Code.

10 · Common Flame Patterns

Object hierarchy — always start here

```
import flame

p  = flame.projects.current_project      # PyProject
ws = p.current_workspace                 # PyWorkspace
desk = ws.desktop                       # PyDesktop

# Libraries are on workspace, NOT on project
libs = ws.libraries                     # list[PyLibrary]

# Reels, clips, folders inside a library
lib = libs[0]
reels = lib.reels                       # list[PyReel]
clips = lib.clips                       # list[PyClip] (direct clips)
folders = lib.folders                   # list[PyFolder]
```

Delete objects — flame.delete()

`flame.delete()` works on any Media Panel object: clips, reels, folders, libraries, sequences, batch groups. Always wrap in a list for multiple objects.

```
import flame
ws = flame.projects.current_project.current_workspace

# Delete a reel by name
lib = next(l for l in ws.libraries if l.name == "Default Library")
reel = next(r for r in lib.reels if r.name == "OLD_REEL")
flame.delete(reel)

# Delete multiple reels at once
targets = {"TEST", "TEST2", "DESKTOP_TEST"}
to_del = [r for r in lib.reels if r.name in targets]
flame.delete(to_del)
print(f"Deleted: {[r.name for r in to_del]}")

# Delete a folder by name
folder = next(f for f in lib.folders if f.name == "OLD_FOLDER")
flame.delete(folder)

# Delete a library
old_lib = next(l for l in ws.libraries if l.name == "TEMP_LIB")
flame.delete(old_lib)

# Delete all clips inside a reel
reel = next(r for r in lib.reels if r.name == "DAILIES")
flame.delete(list(reel.clips))
print(f"Cleared {len(reel.clips)} clips")
```

Create library / reel

```
import flame
ws = flame.projects.current_project.current_workspace
lib = ws.create_library("Incoming")
reel = lib.create_reel("Raw")
print(f"Created: {lib.name} / {reel.name}")
```

List libraries and reels

```
import flame
ws = flame.projects.current_project.current_workspace
for lib in ws.libraries:
    print(f"[{lib.name}]  ({len(lib.reels)} reels, {len(lib.clips)} clips)")
    for reel in lib.reels:
        print(f"  Reel: {reel.name}  ({len(reel.clips)} clips)")
    for folder in lib.folders:
        print(f"  Folder: {folder.name}")
```

Import media

```
import flame
ws = flame.projects.current_project.current_workspace
lib = next(l for l in ws.libraries if l.name == "Default Library")
reel = next(r for r in lib.reels if r.name == "Incoming")
clips = flame.import_clips("/path/to/file.mov", reel)
print(f"Imported: {[c.name for c in clips]}")
```

Render via Background Reactor (non-blocking)

```
import flame

def do_render():
    flame.batch.render(render_option="Background Reactor")
    print("Render queued")

flame.schedule_idle_event(do_render)
print("Render scheduled")
```

Never call `flame.batch.render()` directly — it blocks and can crash Flame's UI.

11 · Logs

Log file	Location	Contents
Bridge log	logs/flame_mcp_bridge.log	Every TCP connection: EXEC <first line of code>, → ok output: <first 80 chars>, or → ERROR
RAG log	logs/flame_rag.log	Every search query, top-3 results with relevance %, char count, token estimate. ERROR entries

Bridge log example

```
[11:50:20] EXEC: import flame
[11:50:20]   → ok  output: "Deleted: ['TEST', 'TEST2']"
[11:51:36]   → ERROR: StopIteration
[11:52:18] Reload: reloading 'flame_mcp_bridge'
[11:52:18] Reload: done — open the menu again to see changes
[11:49:32] Chat widget error: 'PySide6.QtCore.QObject.installEventFilter' called
           with wrong argument types: installEventFilter(_EnterCatcher)
```

RAG log example

```
[10:52:35] Index loaded — 46 chunks
[10:52:35] QUERY: 'create reel in library'
[10:52:35]   → [59%] FLAME_API.md :: Pattern: create library + reel + import
[10:52:35]   → [58%] FLAME_API.md :: Find a reel inside a library
[10:52:35]   → returned 3 chunks, ~813 chars (~203 tokens saved vs full doc)
[11:50:16] QUERY: 'delete object flame.delete'
[11:50:16]   → [47%] FLAME_API.md :: Utility Functions      ← bad match
[11:50:16]   → [44%] FLAME_API.md :: Dialog (blocking)      ← irrelevant
[11:50:16]   → returned 3 chunks, ~1196 chars
```

Low-score entries in the RAG log (< 60%) identify gaps in FLAME_API.md. After learn_pattern runs, these entries will show high scores (>70%) in future sessions.

12 · Troubleshooting

Problem	Cause	Solution
Claude can't connect to Flame	Bridge not running, or Flame not installed.	1. Open Flame. 2. Check MCP Bridge → Status. 3. Verify hook is in /opt/Autodesk/shared/python/. 4. Run: <code>ls -i :4444</code> (should show Flame listening).
Low RAG scores (< 60%) on common operations	Pattern not documented in FLAME_API.md	1. Run <code>python rag/build_index.py</code> after a successful run. Or manually: <code>python rag/build_index.py</code> after editing FLAME_API.md.
Claude Chat doesn't open	ANTHROPIC_API_KEY missing	1. Check logs/flame_mcp_bridge.log for full traceback. 2. Set ANTHROPIC_API_KEY in environment or .env file. 3. Reload hook after updating the bridge.
Hook changes not taking effect	Flame still running old version of hook	1. Run: <code>cp hooks/flame_mcp_bridge.py /opt/Autodesk/shared/python/</code> Then: MCP Bridge → Reload hook (no Flame restart needed).
Terminal launch fails (oh-my-zsh)	oh-my-zsh update prompt intercepts Claude Chat prompt	1. Use Claude Chat (doesn't need a terminal). Or add <code>DISABLE_AUTO_UPDATE=true</code> to <code>~/.zshenv</code> manually.
Port 4444 in use	Another process is using port 4444	1. Change BRIDGE_PORT in both flame_mcp_bridge.py and flame_mcp_server.py. Values must match. Rebuild and redeploy both files.
pip install fails (--user conflict)	Global pip config has install.user = true	Add --no-user flag: <code>pip install -r requirements.txt --no-user</code>
StopIteration in bridge log	next() call on empty iterator (e.g. <code>list_of_names = []</code>)	1. Add a more failback: <code>next((x for x in col if x.name == 'NAME'), None)</code> and check for None before calling <code>flame.delete()</code> .
RAG index not found error	Index hasn't been built, or rag/index/ProjectName	1. Run: <code>cd ~/ProjectName/flame-mcp && source .venv/bin/activate</code> <code>python rag/build_index.py</code>

13 · Compatibility

Flame version	Internal Python	Qt binding	Chat widget	Status
2023	3.9.7	PySide2	✓	✓ Compatible
2024	3.9.x	PySide2	✓	✓ Compatible
2025	3.11.x	PySide2	✓	✓ Compatible
2026	3.11.5	PySide6	✓	✓ Tested
2027 preview	3.13.3	PySide6	✓	✓ Compatible

Claude client compatibility

Client	Connection	Works?	Notes
Claude Code (terminal)	MCP stdio	✓ Full	Primary context. CLAUDE.md provides additional instructions.
Claude Desktop	MCP stdio	✓ Full	Configure via claude_desktop_config.json. Workflow rules from FastMCP instruction
Cowork	MCP stdio	✓ Full	Same server, same tools, same behaviour. No extra config needed.
Embedded Chat widget	Direct API	✓ Full	Calls Anthropic API directly via urllib. Separate from MCP. Works offline from Claude

Project structure

```
flame-mcp/
├── flame_mcp_server.py      # MCP server: 8 tools, token tracking, learn_pattern
├── hooks/
│   ├── flame_mcp_bridge.py  # Flame hook: TCP bridge + Qt chat + menu
│   └── rag/
│       ├── build_index.py   # Build ChromaDB index from FLAME_API.md
│       ├── search.py        # search(query) → (text, max_relevance_int)
│       └── index/           # ChromaDB store (git-ignored, rebuild locally)
├── FLAME_API.md            # Flame API patterns + auto-learned entries
├── CLAUDE.md               # Rules for Claude Code terminal context
├── claude_desktop_config.json # Claude Desktop MCP configuration template
├── requirements.txt         # mcp, chromadb, sentence-transformers
├── install.sh              # Automatic installer
├── LICENSE                 # MIT
├── logs/
│   ├── flame_mcp_bridge.log  # TCP activity: EXEC, errors, reloads
│   └── flame_rag.log         # RAG queries, scores, token savings
└── docs/
    └── flame-mcp-reference.pdf # This document
```