Eclipse-Agent-Plugin

Copyright © 2026 synonet. Alle Rechte vorbehalten.

Ein Eclipse-Plugin, das einen autonomen LLM-Agenten direkt in die IDE bringt. Der Agent steuert den Workspace über die Eclipse-API (Ressourcen, JDT-Build, JUnit) und kann zusätzlich die Oberfläche bedienen (Befehle, Views, Perspektiven, Einstellungen) — ohne Shell, ohne Pixel-Robot.

Dieses Dokument beschreibt **Installation** und **Bedienung** des Plugins. Ausführliche technische Details, Pitfalls und Test-Rezepte liegen im Projekt selbst (Ordner `docs/`, `tests/`, `references/` im Skill).

---

1. Voraussetzungen

Voraussetzung	Wert
Eclipse	Java-Paket **2024-06 R** (getestete Target-Platform)
Java (für das Plugin)	**JDK 21** (Bundle verlangt JavaSE-21)
LLM-Endpoint	OpenAI-kompatibler Endpoint (`/v1`) — **kein Default**, muss vor der ersten Nutzung konfiguriert werden
Betriebssystem	Linux oder Windows (beides verifiziert)

Das Plugin bündelt keinen eigenen Model-Server und bringt **keinen vorkonfigurierten Endpoint** mit. Es spricht einen beliebigen OpenAI-kompatiblen Endpoint an (vLLM, OpenRouter, ollama, ...); die Base-URL wird vom Anwender unter *Preferences → Eclipse-Agent* eingetragen. Ohne konfigurierten, erreichbaren Endpoint funktionieren die integrierten Selbsttests (Mock-Modus), aber keine echten Agenten-Antworten — das Plugin zeigt dann eine klare Hinweismeldung.

---

2. Installation

Es gibt zwei Wege: aus dem Quellcode bauen (empfohlen für die Weiterentwicklung) oder ein fertig gebautes Jar installieren.

2.1 Aus dem Quellcode bauen

```
cd /pfad/zu/eclipse-agent
./build.sh                # baut out/de.synonet.eclipseagent_0.1.0.jar
./scripts/deploy.sh       # kopiert das Jar nach plugins/ UND dropins/
```

Wichtig (aus dem Build-Rezept):

- Das Jar muss **immer in beide Verzeichnisse** der Eclipse-Installation:
- `<eclipse>/plugins/de.synonet.eclipseagent_0.1.0.jar`

- `<eclipse>/dropins/de.synonet.eclipseagent_0.1.0.jar` Nur `plugins/` reicht bei der EPP-Distribution nicht (simpleconfigurator ignoriert fremde Jars; nur der `dropins`-Mechanismus lädt sie zuverlässig).
- Nach jedem Jar-Update Eclipse **mit `-clean`** starten. Das Bundle hat bewusst keine Versions-Bumps (immer `0.1.0`); ohne `-clean` serviert OSGi die alte `plugin.xml` und neue Extension-Punkte erscheinen nicht.

2.2 Fertiges Jar installieren (z. B. auf einem anderen Rechner)

1. Das Jar aus dem ZIP entpacken: `eclipse-agent-plugin_0.1.0/de.synonet.eclipseagent_0.1.0.jar`
2. In plugins/ **und** dropins/ des Ziel-Eclipse kopieren.
3. Eclipse mit `-clean` starten (unter Windows: `eclipse.exe ... -clean`).

Beispiel unter Windows:

```
copy /Y /B C:\tmp\de.synonet.eclipseagent_0.1.0.jar D:\eclipse-agent-test\eclipse\plugins\
copy /Y /B C:\tmp\de.synonet.eclipseagent_0.1.0.jar D:\eclipse-agent-test\eclipse\dropins\
```

Prüfen, ob das Jar korrekt installiert ist, per MD5 (beide Pfade gleich):

```
md5sum target-platform/eclipse/plugins/de.synonet.eclipseagent_0.1.0.jar \
       target-platform/eclipse/dropins/de.synonet.eclipseagent_0.1.0.jar
```

2.3 Installation verifizieren

Startet man Eclipse nach der Installation, erscheint:

- in der Menüleiste ein neues Menü **„Agent"** mit dem Eintrag **„Chat öffnen"**,
- im Java-Perspective unten (View-Stack neben Problems/Console) die Ansicht **„Eclipse-Agent Chat"**,
- unter *Window → Preferences* eine neue Seite **„Eclipse-Agent"**.

Zusätzlich können die integrierten Selbsttests headless laufen (s. 4.4).

---

3. Konfiguration (Endpoint / Modell / Schreibschutz)

Alle Einstellungen liegen in den Eclipse-Preferences (pro Workspace) und sind über *Window → Preferences → Eclipse-Agent* sichtbar und editierbar. Standardwerte:

Präferenz	Key	Default
Base-URL (Endpoint)	`eclipseagent.baseUrl`	*(leer — Pflicht, vor der ersten Nutzung zu setzen)*
Modell	`eclipseagent.model`	*(leer — Pflicht, Modellname des Endpoints)*
API-Key	`eclipseagent.apiKey`	*(leer)*
Maximale Iterationen	`eclipseagent.maxIter`	`25`

Präferenz	Key	Default
Timeout pro LLM-Call [s]	`eclipseagent.timeoutSec`	`300`
Schreibschutz-Gate	`eclipseagent.allowAutoApply`	`false`
Editor-Auswahl anhängen	`eclipseagent.attachSelection`	`true`

Wichtig: Es ist absichtlich **kein** Default-Endpoint und **kein** Modell vorkonfiguriert — das Plugin wird neutral ausgeliefert. Vor der ersten echten Nutzung muss unter *Window → Preferences → Eclipse-Agent* die **Base-URL** des eigenen, OpenAI-kompatiblen Endpoints UND der dort verfügbare **Modellname** eingetragen werden (ggf. zusätzlich ein API-Key, falls der Endpoint einen verlangt). Solange Base-URL oder Modell leer sind, bricht ein Agent-Lauf mit einer klaren Meldung ab und verweist auf diese Einstellungsseite.

Schreibschutz-Gate — das wichtigste Sicherheitsfeature: Ohne `allowAutoApply=true` darf der Agent nicht schreiben. `file_apply_patch` liefert dann nur die Diff-Vorschau (`written:false`) und ändert nichts auf der Platte. Erst wenn das Gate offen ist, werden Änderungen wirklich angewandt. Im Bridge-/Headless-Betrieb steuert dasselbe Flag die Property `-Dagent.bridge.autoApply=true`.

---

4. Bedienung

4.1 Chat-Ansicht (GUI, der normale Weg)

1. Menü **Agent → Chat öffnen** oder Tastenkürzel **Strg+Shift+Alt+A**. Die Ansicht „Eclipse-Agent Chat" öffnet sich; das Eingabefeld hat den Fokus.
2. Aufgabe eintippen und mit **Enter** senden (Shift+Enter = Zeilenumbruch). Der Agent läuft in einem Hintergrund-Job; Tool-Aufrufe und die finalen Antworten erscheinen live im Verlauf.
3. Während eines Laufs: **Stopp** bricht den Agenten ab (wirkt zwischen Modell-Antworten/Tool-Aufrufen). Symbole in der View-Toolbar: *Stopp*, *Leeren*, *Neue Unterhaltung*.
4. Die Unterhaltung bleibt über mehrere Sends erhalten (Session-Kontext).

Editor-Auswahl wird automatisch mitgesendet: Ist im aktiven Editor Text markiert, hängt die Ansicht die Markierung automatisch als Kontext an die Aufgabe (Zeile „[Editor-Auswahl beigefügt: ...]"). Der Agent „sieht" also den markierten Code. Abschaltbar über die Präferenz `eclipseagent.attachSelection`.

Transkripte: Jeder Lauf wird als Textdatei protokolliert: `<workspace>/.metadata/.plugins/de.synonet.eclipseagent/transcripts/run-<zeitstempel>.txt` (dort stehen Log, Tool-Aufrufe und Finalantwort).

4.2 Was der Agent kann (Werkzeuge)

Der Agent hat Zugriff auf Werkzeuge (Tools) — keine Shell. Im Werkzeug-Referenz (s. Kapitel 6) stehen alle. Grob eingeteilt:

- **Workspace:** `workspace_list` (Projekte/Dateien ansehen), `file_read`, `file_apply_patch` (editieren/neu anlegen, hinter dem Schreib-Gate), `build_workspace` (JDT-Build), `run_junit` (Tests ausführen).

- **IDE/UI:** Befehle ausführen, Views/Perspektiven öffnen, Einstellungen lesen und setzen, Editor öffnen, Editor-Zustand abfragen, Dialoge schließen — all das nur in der laufenden GUI (headless nicht verfügbar).
- Der Agent soll **keine** Command-IDs raten: `ide_list_commands` und `ide_find_command` lösen Menü-Labels in IDs auf.

4.3 Fernsteuerung per Bridge (REST/JSON)

Neben der GUI kann derselbe Werkzeugsatz über eine kleine HTTP-Bridge bedient werden — z. B. aus Skripten oder eigenen Agenten, ohne die Oberfläche anzufassen. Zwei Betriebsarten:

a) Headless-Bridge (kein UI, eigener Workspace):

```
./scripts/run_bridge.sh                       # Default: 127.0.0.1:8471
# mit Token + offenem Schreib-Gate:
AGENT_BRIDGE_TOKEN=geheim AGENT_BRIDGE_AUTO_APPLY=1 ./scripts/run_bridge.sh
```

b) Bridge in der laufenden GUI (deterministisches Bedienen der Oberfläche): Eclipse mit der Property starten, dann ist der HTTP-Server in der Workbench aktiv:

```
eclipse -data <workspace> -clean -vmargs \
  -Dagent.bridge.inGui=true -Dagent.bridge.port=8471 -Dagent.bridge.token=e2etok
```

Routen: `GET /bridge/ping` (Health), `GET /bridge/info` (Workspace + Tool-Schemas), `POST /bridge/tool` (ein Tool ausführen), `POST /bridge/task` (kompletter Agenten-Lauf), `POST /bridge/shutdown` (beendet die Bridge, nicht die IDE).

CLI-Client (Python, deterministisches JSON):

```
./scripts/bridge_client.py ping
./scripts/bridge_client.py info
./scripts/bridge_client.py tools
./scripts/bridge_client.py tool workspace_list '{"project":"demo"}'
./scripts/bridge_client.py task "Erkläre kurz die Klasse Hello.java" --max-iter 6
./scripts/bridge_client.py shutdown
```

Optionen (positionsunabhängig): `--host` (Default 127.0.0.1), `--port` (Default 8471), `--token <...>`, `--timeout <sek>`. Bei Token-Pflicht liefert jede Route ohne korrekten Token `401`.

4.4 Headless-Selbsttests (Installation prüfen ohne Netz)

```
AGENT_SELFTEST=1    ./scripts/run_headless.sh   # Mock: workspace_list + Schreib-Gate
AGENT_APPLY_TEST=1  ./scripts/run_headless.sh   # Mock: diff-vorschau + echtes Schreiben
AGENT_JUNIT_TEST=1  ./scripts/run_headless.sh   # Mock: echter run_junit-Lauf im Fixture
./scripts/run_headless.sh                       # echter Lauf gegen den konfigurierten Endpoint
```

Erwartete Ausgaben (Verdicts stehen im Konsole-Log): - Selftest: Gate offen/zu korrekt, workspace_list liefert `demo`. - Apply-Test: Patch wird geschrieben, `written:true`. - JUnit-Test:

`classes_run=2, tests_run=3, failures=1 (HelloFailTest), ignored=1` — der eine Rotlauf ist Absicht (bewusst fehlschlagender Test im Fixture).

---

5. Beispiele

Alle Beispiele nutzen das mitgelieferte Java-Projekt **`demo`** (Klassen `Hello`, `Calc` mit `add(a,b)`, Tests `CalcTest` / `HelloFailTest`) — kein SQL-Beispiel.

Beispiel 1 — Codes zusammenfassen (GUI)

1. Strg+Shift+Alt+A (Chat öffnen).
2. Tippen: `Fasse die Klasse Hello.java kurz zusammen.`
3. Enter.

Antwort des Agenten (Beispiel):

> Hello.java ist eine kleine Demonstrations-Klasse mit einer `greeting()`-Methode, die den Text „Hello, Eclipse-Agent!" zurückgibt. Die Klasse hat keine Abhängigkeiten und keine Nebenwirkungen — sie dient als Trivial-Beispiel für den Workspace-Test.

Beispiel 2 — markierten Code erklären (GUI, Editor-Auswahl)

1. `Calc.java` im Editor öffnen, die Methode `add` markieren.
2. Menü Agent → Chat öffnen.
3. Tippen: `Beantworte nur kurz: Was macht die markierte Methode?`
4. Enter.

Im Verlauf erscheint die graue Zeile `[Editor-Auswahl beigefügt: ...Calc.java, Zeilen 1-8]`; der Agent antwortet ohne Dateizugriff allein aus der Markierung:

> Die markierte Methode `add(int a, int b)` addiert ihre beiden Parameter und gibt das Ergebnis als `int` zurück.

Beispiel 3 — Bridge: Command-ID finden und ausführen

```
# Menü-Label → Command-ID (der Agent muss IDs nie raten):
./scripts/bridge_client.py tool ide_find_command '{"name":"Preferences"}'
# → org.eclipse.ui.window.preferences

./scripts/bridge_client.py tool ide_list_commands '{"filter":"save"}'
# → enthält Save / Save All / Save As (mit IDs + Keybindings)

# Befehl direkt ausführen:
./scripts/bridge_client.py tool ide_execute_command '{"id":"org.eclipse.ui.file.saveAll"}'
# → ok:true
```

Beispiel 4 — Bridge: komplette Agenten-Task

```
./scripts/bridge_client.py task \
  "Schaue dir demo an: Liste die Java-Dateien, baue das Projekt und führe die JUnit-Tests aus. F
  --max-iter 12
```

Der Agent ruft dabei selbstständig `workspace_list` → `file_read` → `build_workspace` → `run_junit` auf und liefert eine deutsche Zusammenfassung (Tests grün/rot, Anzahl Klassen, Laufzeit).

Beispiel 5 — Bridge: Theme/Einstellung setzen (GUI-Bridge)

Voraussetzung: Eclipse ist mit `-Dagent.bridge.inGui=true` gestartet.

```
./scripts/bridge_client.py tool ide_set_preference \
  '{"node":"org.eclipse.e4.ui.css.swt.theme","key":"themeid","value":"org.eclipse.e4.ui.css.them
# → live_applied:true, old_value wird zurückgegeben

# Zurück auf hell (Undo-Beispiel):
./scripts/bridge_client.py tool ide_set_preference \
  '{"node":"org.eclipse.e4.ui.css.swt.theme","key":"themeid","value":"org.eclipse.e4.ui.css.them
```

---

6. Werkzeug-Referenz (alle Tools)

Tool	Funktion	Verfügbar
`workspace_list`	Projekte/Dateien im Workspace auflisten	Headless + GUI
`file_read`	Datei lesen (mit Zeilennummern)	Headless + GUI
`file_apply_patch`	Datei editieren oder neu anlegen (Diff-Operationen; Schreib-Gate)	Headless + GUI
`build_workspace`	JDT-Build des Workspace (Fehler/Warnungen)	Headless + GUI
`run_junit`	JUnit-4-Tests ausführen (Klassen-/Testzahlen, Failures)	Headless + GUI
`ide_problems`	Problem-Marker anzeigen (filterbar)	GUI
`ide_execute_command`	registrierte Eclipse-Command-ID ausführen	GUI
`ide_list_commands`	Command-Katalog auflisten (Filter, Parameter, Keybindings)	GUI
`ide_find_command`	Menü-Label → Command-ID (exakt/Fuzzy) + optional ausführen	GUI
`ide_list_views`	Views auflisten	GUI

Tool	Funktion	Verfügbar
`ide_show_view`	View öffnen (per ID oder Label)	GUI
`ide_list_perspectives`	Perspektiven auflisten	GUI
`ide_show_perspective`	Perspektive aktivieren (per ID oder Label)	GUI
`ide_list_preference_pages`	Preference-Seiten auflisten (Filter)	GUI
`ide_open_preference_page`	Preference-Seite öffnen	GUI
`ide_get_preference`	Präferenzwert lesen (Node/Key)	GUI
`ide_set_preference`	Präferenz setzen (liefert alten Wert; Theme-Wechsel live)	GUI
`ide_open_editor`	Datei im Editor öffnen (optional Zeilensprung)	GUI
`ide_editor_snapshot`	aktiver Editor + offene Editoren (dirty, Cursor-Zeile)	GUI
`ide_show_message`	Info-/Warn-/Fehler-Dialog an den Benutzer	GUI
`ide_close_dialogs`	alle offenen SWT-Dialoge schließen (Hauptfenster bleibt)	GUI

`file_apply_patch` — Operationen: `create` (neue Datei), `replace` (`find`/`replace`), `insert`, `delete`, `append`. Beispiel:

```
{"path":"demo/src/demo/Neu.java",
 "operations":[{"op":"create","text":"package demo;\npublic class Neu {}\n"}],
 "apply":true}
```

---

7. Einrichtung eines eigenen LLM-Endpoints

Jeder OpenAI-kompatible `/v1`-Endpoint funktioniert. In *Window → Preferences → Eclipse-Agent*:

- **Base-URL**: `http://<host>:<port>/v1` — z. B. für eine lokale vLLM-Rig `http://127.0.0.1:8001/v1`, für ollama `http://127.0.0.1:11434/v1`.
- **Modell**: der beim Endpoint verfügbare Modellname.
- **API-Key**: bei Bedarf (z. B. öffentliche Provider); leer bei Haus-Rigs.

Danach neue Unterhaltung starten und eine kurze Testfrage senden.