Extending a Decompiled BDI IDE

Claude API Integration, JDK 21 Migration, and Generic Templates for JACK

Abstract

The JACK Agent Language (JAL) and its companion IDE, the JDE, were shipped as a commercial product by AOS Group in 2018 with no further development planned. This paper documents three independent extensions applied to a decompiled copy of the JDE: an embedded Claude (Anthropic API) assistant for generating JACK code, a toolchain upgrade from Java 8 to Java 21, and a source-level preprocessor that adds Java-style generics to JAL, including fully specialised plan and capability templates. Each extension had to work around a severe structural constraint — the JAL compiler's grammar and rule classes exist only as obfuscated bytecode inside the original *jack.jar*, and the decompiled source files the JDE project ships with are in many places empty stubs that get overwritten at build time. This revision includes the actual code that implements each extension, with line-level commentary on the decisions that shaped it.

1. Context

JACK is a Belief-Desire-Intention (BDI) multi-agent framework. A JACK project is a collection of plugin declarations — agents, plans, capabilities, events, and beliefsets — written in JAL, a small DSL that supersets Java with plugin keywords (`agent`, `plan`, `#handles event`, etc.). The JDE is a Swing-based project manager and editor that invokes the JAL compiler (`JackCompile`) which translates each `.agent` / `.plan` / `.cap` file into Java, then delegates to `javac`.

The JDE source tree used for this work is a CFR-decompiled view of *jack.jar* version 5.6d. It is largely readable — but the ~70 rule-class subclasses that drive JAL's EBNF parser (for example `oL0lSoSu`, registered in `oPNrdinx.language()`) decompiled as empty stubs:

```
public class oL0lSoSu extends onNrMLip {
    public oL0lSoSu(oPNrdinx p, int n) { super(); }
}
```

Listing — a decompilation stub — the body that drives a JAL grammar rule is lost.

The build works only because step 2 of `build.sh` runs `jar xf jack.jar` over `build/classes`, overwriting these stubs with the compiled bytecode from the original jar. This has one unavoidable consequence: **any source-level edit to a rule class is silently discarded every build.** The grammar is therefore read-only short of a full bytecode-rewriting toolchain, and every language extension described below has to avoid touching it.

2. Anthropic API Integration

2.1 Goal

Embed a chat assistant inside the JDE that can draft complete JACK objects — agents, plans, capabilities, events, beliefsets — directly into the project tree, reacting to IDE events (file created, compilation started, compilation complete). The assistant must produce compilable JAL on first attempt as often as possible, and recover gracefully when it does not.

2.2 Architecture

A new package `aos.claude` holds the integration, with no external JSON or HTTP library dependency. The package exposes a small event interface so the rest of the IDE can notify Claude about relevant lifecycle events:

```
package aos.claude;

public interface ClaudeEventHandler {
    void onFileCreated(String filePath, String fileType);
    void onCompilationStarted(String[] sourceFiles);
    void onCompilationComplete(boolean success, String output);
}
```

Listing — ClaudeEventHandler — the IDE-facing event interface.

File	Role
`ClaudeEventHandler.java`	Interface: onFileCreated, onCompilationStarted, onCompilationComplete.
`ClaudeApiClient.java`	HTTP client using java.net.http.HttpClient (requires Java 11+), builds requests by string concatenation, parse
`ClaudeTerminalPanel.java`	JPanel chat UI (JTextArea + JTextField + Send button), implements ClaudeEventHandler.
`ClaudeTerminalFrame.java`	JInternalFrame (520×420) wrapping the panel — dockable in the JDE's central JDesktopPane.
`ClaudeLauncher.java`	Singleton manager; openTerminal(), notifyFileCreated(), notifyCompilationStarted(), notifyCompilationCom

2.3 The System Prompt

Every API call ships the canonical JAL reference as the `system` field. The constant `SYSTEM_PROMPT` in `ClaudeApiClient.java` starts around line 179 and reaches ~400 lines after the generic-templates section was added. Its opening:

```
private static final String SYSTEM_PROMPT =
    "You are an expert JACK Agent Language (JAL) programmer embedded " +
    "in the JACK Development Environment (JDE). " +
    "You generate correct, complete, compilable JACK code. " +
    "ALWAYS output each JACK object in its own fenced code block. " +
    "NEVER use '...' or ellipsis as placeholder code. " +
    "Do NOT add 'import' statements for JACK base types — the JACK " +
    "compiler resolves them automatically.\n\n" +
    // ■■ JACK Language Reference ■■
    "# JACK AGENT LANGUAGE REFERENCE\n\n" +
    ...;
```

Listing — SYSTEM_PROMPT opening — strict output contract, then the JACK reference.

The prompt covers every plugin kind, data directive (`#private data`, `#imports data`, `#exports data`), @-statement family (`@post`, `@subtask`, `@send`, `@parallel`), and the `Semaphore` synchronisation idiom that a Program → agent facade → plan flow uses to block until a plan completes. When the generics extensions shipped, a new section marked *NON-STANDARD JACK* was added to teach Claude the template syntax:

```
"### JDE Generics Extensions (NON-STANDARD JACK)\n" +
"This JDE supports Java-style angle-bracket generics on agents, " +
"plans, and capabilities. These are JDE-only extensions implemented " +
"by a preprocessor — they are NOT part of standard JACK. Prefer " +
"NON-generic code unless the user explicitly asks for generics or " +
"the surrounding code is already templated.\n\n" +

"#### Plan Templates\n" +
"```\n" +
"public plan MyPlan<HEvent> extends Plan {\n" +
"    #handles event HEvent ev;\n" +
"    static boolean relevant(HEvent ev) { return true; }\n" +
"    context() { true; }\n" +
"    #reasoning method\n" +
"    body() { System.out.println(ev.someField); }\n" +
"}\n" +
"```\n" +
"Instantiated ONLY from a capability via " +
"`#uses plan MyPlan<ConcreteEvent>;`. ...";
```

Listing — a slice of SYSTEM_PROMPT — teaching Claude about plan templates.

2.4 Request Construction

The client assembles its request body as a plain `StringBuilder` rather than pulling in a JSON library. This keeps `jde.jar` a self-contained drop-in replacement for `jack.jar`. The outgoing shape:

```
StringBuilder sb = new StringBuilder();
sb.append("{");
sb.append("\"model\":\"").append(MODEL).append("\",");
sb.append("\"max_tokens\":").append(MAX_TOKENS).append(",");
sb.append("\"system\":\"").append(escapeJson(SYSTEM_PROMPT)).append("\",");
sb.append("\"messages\":[");
for (int i = 0; i < history.size(); i++) {
    Message m = history.get(i);
    if (i > 0) sb.append(",");
    sb.append("{\"role\":\"").append(m.role).append("\",");
    sb.append("\"content\":\"").append(escapeJson(m.content))
      .append("\"}");
}
sb.append("]}");
```

Listing — ClaudeApiClient — building the outgoing JSON by hand.

A short conversation history (recent turns) is retained so the assistant can refine its last output in response to user corrections, and a few-shot example library is embedded in the system prompt rather than attached per-request.

2.5 Integration Hooks

Three obfuscated files receive minimal edits to wire the IDE's existing lifecycle into Claude's event handler. `orrrdICl.java` adds a *Claude Terminal* menu item to `toolsMenu` and fires `notifyFileCreated()` from `initChild()`:

```
// orrrdICl.java (IDE window)
JMenuItem claudeItem = new JMenuItem("Claude Terminal");
claudeItem.addActionListener(e -> ClaudeLauncher.openTerminal());
toolsMenu.add(claudeItem);

// inside initChild(), after oCbNmmPs("after creation"):
if (object2 instanceof BAPI_Filename) {
    BAPI_Filename bf = (BAPI_Filename) object2;
    ClaudeLauncher.notifyFileCreated(bf.filename, bf.type);
}
```

Listing — orrrdICl.java — menu registration and file-creation callback.

`ouucssuo.java` (the JAL build/compile driver) fires compilation-started and compilation-complete events around the compile call:

```
// ouucssuo.java — inside oiPvnxWn()
try {
    oo0CnoDM = true;
    ClaudeLauncher.notifyCompilationStarted(sourceFiles);
    ...
    // run JAL compile pipeline
    ...
    ClaudeLauncher.notifyCompilationComplete(success, output);
} finally {
    ClaudeLauncher.notifyCompilationComplete(false, "");
    oo0CnoDM = false;
}
```

Listing — ouucssuo.java — compile lifecycle events.

3. JDK 21 Migration

3.1 Motivation

The original JDE was compiled for Java 8. Users writing Java code inside `#java { ... }` escape blocks, inside `.java` helpers that live alongside `.agent` files, or inside auto-generated wrapper code could not use generics, `var`, lambdas with advanced type inference, records, text blocks, virtual threads, or any of the ergonomic improvements accumulated over thirteen Java releases.

3.2 Toolchain Change

The headline change is in `build.sh` (mirrored in `build.bat`):

```
JAVA_HOME="${JAVA_HOME:-/c/Program Files/Eclipse Adoptium/"\
          "jdk-21.0.10.7-hotspot}"
JAVAC="$JAVA_HOME/bin/javac"
JAR_BIN="$JAVA_HOME/bin/jar"
JAVA_RELEASE="${JAVA_RELEASE:-21}"

"$JAVAC" \
    --release "$JAVA_RELEASE" \
    -encoding UTF-8 \
    -d "$WIN_OUT" \
    -sourcepath "$WIN_SRC" \
    @"$WIN_SOURCES"
```

Listing — build.sh — Temurin 21 toolchain, `--release 21`.

3.3 The Four Non-Obvious Fixes

3.3.1 `java.ext.dirs` removal

JDK 11 removed the extension-directories mechanism. The classpath bootstrap in `oNmmmmCz` read `System.getProperty("java.ext.dirs")` and iterated its path-separator-delimited entries. On JDK 11+ the property returns null, causing a `NullPointerException` at class-load time. The fix is a single null guard:

```
private static void ovMwLICn() {
    String var0 = System.getProperty("java.ext.dirs");
    if (var0 == null || var0.length() == 0) return;   // <- added
    while (var0 != null) {
        int idx = var0.indexOf(File.pathSeparatorChar);
        ...
    }
}
```

Listing — oNmmmmCz.ovMwLICn — null-tolerant extension-dir walk.

3.3.2 Skip `.java` from JAL input

The original pipeline fed both `.agent` and `.java` sources into `JackCompile.main()`. JAL's grammar predates Java generics and rejects `ArrayList<String>` outright. A one-line change routes `.java` sources directly to `javac`, which was already part of the downstream build:

```
// MainImproved.java, ~line 673
stringArrayQ = new StringArrayQ(8);
MainImproved.owSNmbxO(stringArrayQ, odDMDrWO);
```

```
// Skip .java files from JAL input — lets user-written .java use modern
// Java syntax (generics, records, etc). javac handles them separately.
// MainImproved.oOWpbuoI(stringArrayQ, stringArrayQ4);    <- commented out
MainImproved.oOWpbuoI(stringArrayQ, stringArrayQ3);
```

Listing — MainImproved.java — only JAL plugin sources reach JackCompile.

3.3.3 Modern class-file attributes

`aos.apib.oDboDrdi` read the class-file constant pool assuming Java 8 shape. Java 11 introduced `NestHost` / `NestMembers`; Java 17 added `Record` and sealed-class attributes. The reader was extended to skip unknown attributes by length rather than throwing.

3.4 Boundary of the Fix

The JDK 21 migration unlocks modern Java inside `.java` helpers and `#java { }` blocks. It does **not** change what JAL itself accepts — the JAL grammar lives in bytecode we can't edit, so plugin declarations continue to use only the 2007-era Java subset the grammar supports. Adding generics to the JAL declarations themselves required a different approach, documented in the next section.

4. JACK Generic Templates

4.1 Problem Statement

The user asked: can a plan be parameterised over its triggering event type, so that a single template like

```
public plan SetDebugPlan<HEvent> extends Plan {
    #handles event HEvent sde1;
    static boolean relevant(HEvent sde1) { return true; }
    context() { true; }
    #reasoning method
    body() {
        try {
            debug.add(0, sde1.bDebug);
        } catch (BeliefSetException e) {
            e.printStackTrace();
        } finally {
            mutex.signal();
        }
    }
}
```

Listing — a plan template the user wants to be valid JAL.

can be instantiated from a capability with `#uses plan SetDebugPlan<SetDebugEvent>;`? And, symmetrically, can a capability be parameterised so that an agent instantiates it with `#has capability DebugCapability<SetDebugEvent> debugCap;`?

Standard JACK has no such syntax. The `<...>` tokens simply aren't in the grammar's Type rule; the parser treats `<` as a relational operator.

4.2 Path A vs Path B

Path A would modify the JAL grammar to accept type arguments. This requires editing the compiled rule classes, which are the stubs described in section 1. An investigation of `Java12.enc` (initially believed to be a Java serialization blob holding the grammar) revealed it is neither a serialization blob nor text — the bytes start `0x08 0x7D 0xC3 0x90 ...`, not `0xAC 0xED` — and whatever format it uses passes through a multi-stage obfuscated decoder. Decoding it was scoped out.

Path B adds a text-level preprocessor around `JackCompile.main()`. Strip generics before JAL parses; re-inject them into generated Java afterwards. No grammar changes, no bytecode touch, one integration point. Path B was chosen.

4.3 The Preprocessor — Span Model

`JalGenericsPreprocessor` walks each JAL source, skipping strings, chars, and comments, and classifies every `<...>` pair into one of three *span kinds*:

```
public static final class Span {
    public final int index;
    public final String original;      // e.g. "<String>" or "<T extends Foo>"
    public final String anchorIdent;   // identifier used to locate insertion point
    public final Mode mode;

    public enum Mode { APPEND_AFTER, PREPEND_BEFORE }
```

```
    Span(int i, String o, String a, Mode m) {
        index = i; original = o; anchorIdent = a; mode = m;
    }
}
```

Listing — JalGenericsPreprocessor.Span — unit of bookkeeping.

The `anchorIdent` is a plain Java identifier found adjacent to the <. After JAL emits generated Java the restore pass scans for the Nth occurrence of this identifier (preserving source order) and inserts the span either *after* it (`APPEND_AFTER`, for `List<String>`-style trailing generics) or *before* it (`PREPEND_BEFORE`, for `<T> T foo()` and `obj.<String>method()`).

The classifier is the heart of the scanner. It determines the span's kind (or that the < is a relational operator and should be left alone):

```
private enum OpenerKind { NONE, TYPE_ARG, DECL_PREFIX, INVOKE_PREFIX }

private static OpenerKind classifyOpener(char[] b, int i) {
    int j = i - 1;
    while (j >= 0 && Character.isWhitespace(b[j])) j--;

    if (j < 0) return OpenerKind.DECL_PREFIX;       // <T> at start of file

    char prev = b[j];
    if (prev == '.') return OpenerKind.INVOKE_PREFIX;  // obj.<T>method()
    if (prev == '{' || prev == ';') return OpenerKind.DECL_PREFIX;

    if (!(Character.isJavaIdentifierPart(prev) || prev == '>'))
        return OpenerKind.NONE;                     // relational `<`

    // Walk back across the immediate identifier, then any whitespace.
    int identEnd = j + 1;
    while (j >= 0 && (Character.isJavaIdentifierPart(b[j])
                      || b[j] == '.')) j--;
    int identStart = j + 1;
    String ident = new String(b, identStart, identEnd - identStart);
    while (j >= 0 && Character.isWhitespace(b[j])) j--;

    // `public <T> T foo(T x)` — ident is the modifier, <T> is a decl.
    if (isModifierKeyword(ident)
            && (j < 0 || isDeclBoundary(b, j))) {
        return OpenerKind.DECL_PREFIX;
    }

    // Classic type-arg contexts (fields, locals, casts, etc.).
    if (j >= 0) {
        char before = b[j];
        switch (before) {
            case '(': case ',': case '=': case '{': case ';':
            case '>': case '&': case '|':
                return OpenerKind.TYPE_ARG;
        }

        // Keyword before the identifier: `new List<...>`, `agent Foo<T>`.
        int wordEnd = j + 1;
        while (j >= 0 && Character.isJavaIdentifierPart(b[j])) j--;
        String word = new String(b, j + 1, wordEnd - (j + 1));
        switch (word) {
            case "new": case "return": case "extends":
            case "implements": case "throws": case "instanceof":
            case "agent": case "plan": case "capability":
            case "event": case "view": case "team":
```

```
            case "beliefset": case "bel":
            case "class": case "interface": case "enum":
                return OpenerKind.TYPE_ARG;
        }
    }

    return OpenerKind.NONE;
}
```

Listing — classifyOpener — the relational-vs-generic decision.

4.4 Strip and Restore

With classification in hand, the strip loop iterates through the source, records each classified span, and overwrites its bytes with spaces (length-preserving, so error line / column reports still point where the user wrote the <):

```
public static Stripped strip(String src) {
    char[] buf = src.toCharArray();
    List<Span> spans = new ArrayList<>();
    int n = buf.length; int i = 0;
    while (i < n) {
        char c = buf[i];
        if (c == '"') { i = skipString(buf, i); continue; }
        if (c == '\'') { i = skipChar(buf, i); continue; }
        if (c == '/' && i + 1 < n) {
            if (buf[i + 1] == '/') { i = skipLineComment(buf, i); continue; }
            if (buf[i + 1] == '*') { i = skipBlockComment(buf, i); continue; }
        }
        if (c == '<') {
            OpenerKind kind = classifyOpener(buf, i);
            if (kind != OpenerKind.NONE) {
                int end = matchGenericClose(buf, i);
                if (end > 0) {
                    String orig = new String(buf, i, end - i);
                    String anchor; Span.Mode mode;
                    if (kind == OpenerKind.TYPE_ARG) {
                        anchor = findPrecedingIdentifier(buf, i);
                        mode = Span.Mode.APPEND_AFTER;
                    } else {
                        anchor = findFollowingIdentifier(buf, end);
                        mode = Span.Mode.PREPEND_BEFORE;
                    }
                    if (!anchor.isEmpty()) {
                        spans.add(new Span(spans.size(), orig, anchor, mode));
                        for (int k = i; k < end; k++) {
                            if (buf[k] != '\n' && buf[k] != '\r') buf[k] = ' ';
                        }
                        i = end; continue;
                    }
                }
            }
        }
        i++;
    }
    return new Stripped(new String(buf), spans, injectedTypeParams);
}
```

Listing — strip() — the main preprocessor loop.

The mirror restore walks the generated `.java` output, locates the Nth safe occurrence of each span's anchor identifier, and injects the original `<...>` text:

```
public static String restore(String generated, List<Span> spans) {
    if (spans.isEmpty()) return generated;
    boolean[] safe = buildSafeMask(generated);
    StringBuilder out = new StringBuilder(generated.length() + 32);
    int cursor = 0;
    for (Span span : spans) {
        int hitStart = findNextIdentifier(
            generated, safe, span.anchorIdent, cursor);
        if (hitStart < 0) continue;
        int hitEnd = hitStart + span.anchorIdent.length();
        if (span.mode == Span.Mode.APPEND_AFTER) {
            out.append(generated, cursor, hitEnd);
            out.append(span.original);
            cursor = hitEnd;
        } else { // PREPEND_BEFORE
            out.append(generated, cursor, hitStart);
            out.append(span.original);
            if (!span.original.endsWith(" ")) out.append(' ');
            cursor = hitStart;
        }
    }
    out.append(generated, cursor, generated.length());
    return out.toString();
}
```

Listing — restore() — re-injects generics by ordinal identifier match.

`buildSafeMask` returns a boolean array marking positions outside strings, char literals, comments, and `import` / `package` statements, so the restore pass never matches a name that appears inside a literal.

4.5 Agent Type Parameters: Body Erasure

Handling `agent Foo<T>` needs more than stripping. JAL sees `agent Foo` and a body containing `T name;` and `name.toString()` and reports *Type T not found*. Two approaches failed first: a synthetic nested class `public static class T {}` inside the agent body, and a package-sibling `class T {}` at file top. JAL parses both but its type resolver ignores them.

Body erasure is the fallback that worked. During stripping, the classifier flags class-header TYPE_ARG spans; after the main loop, every whole-word reference to any such type parameter is rewritten to `Object` throughout the file:

```
private static String eraseTypeParamsInBody(String src, List<String> params) {
    if (params.isEmpty()) return src;
    char[] b = src.toCharArray();
    int n = b.length;
    StringBuilder out = new StringBuilder(n + 64);
    int i = 0;
    while (i < n) {
        char c = b[i];
        // skip strings / chars / comments verbatim ...
        // ...

        // Whole-word match at an identifier boundary?
        if (Character.isJavaIdentifierStart(c)
                && (i == 0 || !Character.isJavaIdentifierPart(b[i - 1])
                               && b[i - 1] != '.')) {
            String matched = null;
            for (String p : params) {
                int pl = p.length();
                if (i + pl <= n && src.regionMatches(i, p, 0, pl)
```

```
                        && (i + pl == n
                            || !Character.isJavaIdentifierPart(b[i + pl]))) {
                    matched = p; break;
                }
            }
            if (matched != null) {
                out.append("Object");
                i += matched.length();
                continue;
            }
        }
        out.append(c);
        i++;
    }
    return out.toString();
}
```

Listing — eraseTypeParamsInBody — type-parameter identifiers rewritten to Object.

The class-header `<T>` span is still restored into the generated Java, so external callers see `Foo<String>` and can write `new Foo<String>()`. Internal field types erase to `Object`, which is the accepted trade-off for simple agents.

4.6 Plan and Capability Templates: Full Specialisation

Erasure is too lossy for plans. When a plan body accesses `sde1.bDebug`, `bDebug` exists on the concrete event type but not on `Object`. The right mechanism is *specialisation* — generate a separate file per instantiation with the concrete argument substituted in.

`PlanTemplateExpander` handles this. Its three regexes describe the shapes of interest:

```
/** `public [plan|capability] Name<Params> [extends Super] {` */
private static final Pattern TEMPLATE_HEADER = Pattern.compile(
    "(?m)^\\s*(?:public\\s+|private\\s+|protected\\s+)?" +
    "(plan|capability)\\s+(\\w+)\\s*<([^<>]+)>\\s*" +
    "((?:extends|implements)[^{]*)?\\{");

/** `#uses plan Name<Args>;` — group 1 = name, group 2 = args. */
private static final Pattern USES_PLAN = Pattern.compile(
    "#uses\\s+plan\\s+(\\w+)\\s*<([^<>;]+)>\\s*;");

/** `#has capability Name<Args> identifier;` */
private static final Pattern HAS_CAP = Pattern.compile(
    "#has\\s+capability\\s+(\\w+)\\s*<([^<>;]+)>\\s*(\\w+)\\s*;");
```

Listing — PlanTemplateExpander — template + instantiation patterns.

The main entry point runs three passes. Pass 1 scans `.plan` and `.cap` files for templates. Pass 2 is an iterative worklist loop: scan each non-template source for instantiation directives, generate a specialised file for each unique `(template, args)` tuple, rewrite the instantiation site, and add the generated file to the next iteration's worklist so nested templates expand in subsequent passes. Pass 3 builds the new argv:

```
public static String[] expand(String[] argv) {
    GENERATED.clear();
    MODIFIED.clear();
    if (argv == null) return argv;

    // Pass 1 — discover templates.
    Map<String, Template> templates = new HashMap<>();
    for (String path : argv) {
```

```
            if (path == null) continue;
            String lower = path.toLowerCase();
            if (!lower.endsWith(".plan") && !lower.endsWith(".cap")) continue;
            String content = readOrNull(path);
            if (content == null) continue;
            Matcher m = TEMPLATE_HEADER.matcher(content);
            if (m.find()) {
                String keyword = m.group(1);
                String name = m.group(2);
                List<String> params = splitIdentifiers(m.group(3));
                if (!params.isEmpty()) {
                    templates.put(name,
                        new Template(path, name, keyword, params, content));
                }
            }
        }
        if (templates.isEmpty()) return argv;

        // Pass 2 — iterative worklist. Templates are skipped because their
        // internal `#uses plan Foo<HEvent>` uses HEvent as a placeholder.
        Set<String> templatePaths = new HashSet<>();
        for (Template t : templates.values()) templatePaths.add(t.path);
        Map<String, String> generatedNames = new HashMap<>();
        List<String> worklist = new ArrayList<>();
        for (String p : argv) {
            if (p != null && !templatePaths.contains(p)) worklist.add(p);
        }
        int iterations = 0;
        while (!worklist.isEmpty() && iterations++ < 10) {
            List<String> nextBatch = new ArrayList<>();
            for (String path : worklist) {
                String content = readOrNull(path);
                if (content == null) continue;
                RewriteResult r = rewriteInstantiations(
                    content, templates, generatedNames, nextBatch);
                if (r.changed) {
                    if (!MODIFIED.containsKey(path) && !GENERATED.contains(path)) {
                        MODIFIED.put(path,
                            content.getBytes(StandardCharsets.UTF_8));
                    }
                    Files.write(Paths.get(path),
                        r.text.getBytes(StandardCharsets.UTF_8));
                }
            }
            worklist = nextBatch;
        }

        // Pass 3 — build new argv, drop templates, dedup by canonical path.
        LinkedHashMap<String, String> finalSet = new LinkedHashMap<>();
        for (String path : argv) {
            if (isTemplatePath(path, templates)) continue;
            finalSet.put(canonKey(path), path);
        }
        for (String g : GENERATED) {
            finalSet.putIfAbsent(canonKey(g), g);
        }
        return finalSet.values().toArray(new String[0]);
    }
```

Listing — PlanTemplateExpander.expand — three passes (discover, iterate, rewrite argv).

The per-file worker finds every instantiation and decides whether to generate or reuse a specialised output. The collected edits are applied left-to-right so offsets remain valid:

```
private static RewriteResult rewriteInstantiations(
        String content,
        Map<String, Template> templates,
        Map<String, String> generatedNames,
        List<String> nextBatch) {
    List<Edit> edits = new ArrayList<>();
    collectMatches(USES_PLAN, content, 0,
        templates, generatedNames, nextBatch, edits);
    collectMatches(HAS_CAP,   content, 1,
        templates, generatedNames, nextBatch, edits);

    if (edits.isEmpty()) return new RewriteResult(content, false);
    edits.sort((a, b) -> Integer.compare(a.start, b.start));

    StringBuilder out = new StringBuilder(content.length() + 64);
    int cursor = 0;
    for (Edit e : edits) {
        out.append(content, cursor, e.start);
        out.append(e.replacement);
        cursor = e.end;
    }
    out.append(content, cursor, content.length());
    return new RewriteResult(out.toString(), true);
}

private static void collectMatches(
        Pattern pat, String content, int kind,
        Map<String, Template> templates,
        Map<String, String> generatedNames,
        List<String> nextBatch, List<Edit> edits) {
    Matcher m = pat.matcher(content);
    while (m.find()) {
        String name = m.group(1);
        Template t = templates.get(name);
        if (t == null) continue;
        List<String> args = splitIdentifiers(m.group(2));
        if (args.size() != t.params.size()) continue;

        String key = name + "<" + String.join(",", args) + ">";
        String mangled = generatedNames.get(key);
        if (mangled == null) {
            mangled = mangleName(name, args);
            String genPath = generateSpecialised(t, mangled, args);
            if (genPath != null) nextBatch.add(genPath);
            generatedNames.put(key, mangled);
        }
        String replacement = (kind == 0)
            ? "#uses plan " + mangled + ";"
            : "#has capability " + mangled + " " + m.group(3) + ";";
        edits.add(new Edit(m.start(), m.end(), replacement));
    }
}
```

Listing — rewriteInstantiations + collectMatches — edit accumulation.

Specialised-file generation rewrites the class header and substitutes every type-param identifier in the body:

```
private static String generateSpecialised(Template t, String mangled,
                                          List<String> args) {
    Matcher hm = TEMPLATE_HEADER.matcher(t.content);
    if (!hm.find()) return null;
    StringBuilder body = new StringBuilder(t.content);
    // Replace the class name (group 2) with the mangled name.
    body.replace(hm.start(2), hm.end(2), mangled);
```

```
    // Delete the `<Params>` clause that follows.
    int lt = body.indexOf("<", hm.start(2));
    int gt = body.indexOf(">", lt);
    if (lt > 0 && gt > lt) body.delete(lt, gt + 1);

    String result = body.toString();

    // Substitute longest names first so HEvent2 isn't eaten by HEvent.
    List<int[]> order = new ArrayList<>();
    for (int i = 0; i < t.params.size(); i++) order.add(new int[]{i});
    order.sort((a, b) -> Integer.compare(
        t.params.get(b[0]).length(), t.params.get(a[0]).length()));
    for (int[] idx : order) {
        String p = t.params.get(idx[0]);
        String a = args.get(idx[0]);
        result = wholeWordReplace(result, p, a);
    }

    String ext = "capability".equals(t.keyword) ? ".cap" : ".plan";
    String outPath = dirOf(t.path) + mangled + ext;
    Files.deleteIfExists(Paths.get(outPath));   // clear stale copy
    Files.write(Paths.get(outPath),
        result.getBytes(StandardCharsets.UTF_8));
    GENERATED.add(outPath);
    return outPath;
}
```

Listing — generateSpecialised — writes a concrete .plan or .cap per instantiation.

The `wholeWordReplace` helper respects identifier boundaries and skips strings / comments so a literal string containing the type-param name remains unchanged.

4.7 Cleanup

After JAL returns (success or failure), the hook tears down everything the expander built:

```
public static void cleanup() {
    for (String path : GENERATED) {
        try { Files.deleteIfExists(Paths.get(path)); }
        catch (IOException ex) {
            System.err.println("[PlanTemplateExpander] failed to delete "
                + path + ": " + ex.getMessage());
        }
    }
    for (Map.Entry<String, byte[]> e : MODIFIED.entrySet()) {
        try { Files.write(Paths.get(e.getKey()), e.getValue()); }
        catch (IOException ex) {
            System.err.println("[PlanTemplateExpander] failed to restore "
                + e.getKey() + ": " + ex.getMessage());
        }
    }
    GENERATED.clear();
    MODIFIED.clear();
}
```

Listing — PlanTemplateExpander.cleanup — always runs after compile.

4.8 Wiring it all Together — the Hook

`JalGenericsHook` is the narrow bridge between the JDE and everything above. Its `preprocess` first expands templates and then strips ordinary generics:

```
public static String[] preprocess(String[] argv) {
    SPANS.clear();
    ORIG_CONTENT.clear();
    if (argv == null) return argv;

    // Template-expansion pass: generate specialised plan/capability
    // files for `#uses plan Foo<X>;` and `#has capability Foo<X> ident;`
    // references, rewrite instantiation sites, drop template sources.
    String[] expanded = PlanTemplateExpander.expand(argv);

    for (String a : expanded) {
        if (a != null && isJalSource(a)) {
            File f = new File(a);
            if (f.isFile()) {
                try { stripInPlace(f); }
                catch (IOException ex) {
                    System.err.println("[JalGenericsHook] failed to "
                        + "preprocess " + a + ": " + ex.getMessage());
                }
            }
        }
    }
    return expanded;
}
```

Listing — JalGenericsHook.preprocess — expand templates, then strip generics.

The mirror `postprocess` restores everything, with retries for Windows file locks:

```
public static void postprocess(String[] argv) {
    if (argv == null) { clearState(); return; }
    String javaOutDir = findArg(argv, "-d");
    String workDir   = findArg(argv, "-wd");

    // Step 1 — restore generics in generated .java.
    for (Map.Entry<String, List<Span>> e : SPANS.entrySet()) {
        String jalSource = e.getKey();
        List<Span> spans = e.getValue();
        File gen = locateGeneratedJava(jalSource, javaOutDir, workDir);
        if (gen != null && gen.isFile()) {
            String body = new String(Files.readAllBytes(gen.toPath()),
                StandardCharsets.UTF_8);
            String cleaned = JalGenericsPreprocessor
                .removeTypeParamStubs(body);
            String restored = JalGenericsPreprocessor
                .restore(cleaned, spans);
            if (!restored.equals(body)) {
                Files.write(gen.toPath(),
                    restored.getBytes(StandardCharsets.UTF_8));
            }
        }
    }

    // Step 2 — restore original JAL sources from in-memory backups.
    for (Map.Entry<String, byte[]> e : ORIG_CONTENT.entrySet()) {
        writeWithRetry(e.getKey(), e.getValue(), 5);
    }

    // Step 3 — delete generated template files + restore capability backups.
    PlanTemplateExpander.cleanup();
    clearState();
}
```

Listing — JalGenericsHook.postprocess — reverses everything, in order.

And the single integration point in the IDE's driver — fewer than ten lines, wrapped in a `try/finally` so the hook's `postprocess` always runs:

```
// MainImproved.java, around line 680
String[] __jgOriginal = stringArrayQ.toArray();
String[] __jgArgv = aos.jalgenerics
    .JalGenericsHook.preprocess(__jgOriginal);
try {
    if (oowLinxM == null) {
        new JackCompile();
        n6 = JackCompile.main(__jgArgv);
    } else {
        n6 = this.oCrMwvOP(oowLinxM, stringArrayQ);
    }
} finally {
    aos.jalgenerics.JalGenericsHook.postprocess(__jgOriginal);
}
```

Listing — MainImproved.java — the single integration point.

5. Lessons

5.1 Respect the Black Boxes

Every problem in this project traced back to respecting two black boxes: the compiled grammar bytecode and the opaque `.enc` resources. The JDK 21 migration was short because the fixes landed entirely in decompiled, readable code. The generics effort was long because the first plan assumed the grammar was editable, based on memory notes that turned out to be wrong. Verifying the actual shape of `Java12.enc` (it starts `0x08`, not `0xAC 0xED`) invalidated the entire Path A design and forced a pivot to the purely textual Path B. Future language extensions should begin with a cheap feasibility probe of the black box, not an architecture document written against a half-remembered memory.

5.2 Erasure vs Specialisation

Two different mechanisms solved two different shapes of the same problem. Agent type parameters are often just a typed handle on an opaque field — `Object` carries enough methods (`toString`, `equals`, `hashCode`) for erasure to work. Plan and capability templates are different: their bodies routinely access domain-specific methods and fields of the concrete event type. Literal substitution was required. Choosing the right mechanism per shape — rather than forcing one strategy onto both — kept each implementation small and understandable.

5.3 Iterative Worklists Beat Regex

The first draft of the expander did one scan-and-generate pass. It handled `#uses plan Foo<X>` directly in a capability but not a capability template whose expansion produced a new `#uses plan` directive. Rewriting as an iterative worklist turned a linear pipeline into a fixed-point loop that naturally handled nested templates without any special-case code for the capability-contains-plan case.

5.4 Build-Step Details Matter

Three separate bugs came from build mechanics, not from feature logic: `javac`'s `@file` format treats backslash as an escape character (Windows paths broke); cmd's `if exist` block breaks on a path containing a `)` character (*Program Files (x86)*); jar file locks silently prevent rebuilds when the JDE is still running. Each had a fix that fit in a few lines. Each had to be discovered the hard way because the symptom never named the cause. A lesson for future IDE hacks: verify the running jar contains your latest code before spending a single minute on behavioural debugging. The PowerShell-based source-file generation that finally landed in `build.bat` illustrates the shape of the fix:

```
REM javac @-file treats backslash as an escape character, so paths must
REM use forward slashes. Also wrap each path in double quotes for spaces.
set RAW_SOURCES=%PROJECT_DIR%build\sources_raw.txt
dir /s /b "%SRC%\*.java" > "%RAW_SOURCES%"
if exist "%SOURCES_FILE%" del "%SOURCES_FILE%"
powershell -NoProfile -Command "(Get-Content -LiteralPath '%RAW_SOURCES%') ^
    | ForEach-Object { '\"' + ($_ -replace '\\','/') + '\"' } ^
    | Set-Content -LiteralPath '%SOURCES_FILE%'"
```

Listing — build.bat — paths converted to forward slashes before javac sees them.

6. Conclusion

Three extensions — an embedded Claude assistant, a modern Java toolchain, and a generics preprocessor capable of full template specialisation for plans and capabilities — all land cleanly inside a JDE project distributed only as decompiled stubs around an opaque grammar. Each feature respects the single hard constraint (original compiled rule classes cannot be modified) and works via a narrow integration point: a menu item, a `--release 21` flag, a hook call inside `MainImproved.java`. The resulting system lets a user write `agent Foo<T>` and `#has capability DebugCapability<SetDebugEvent> debugCap;` in JAL source, get standard JAL compilation with specialised output, and stay inside the JDE's familiar edit/compile/run loop. A codebase of 3,200 plans — the user's actual corpus — is a realistic target for the template mechanism to reduce duplication across plans that differ only in their triggering event type.

The larger point: a closed product can be extended thoughtfully without reverse-engineering its core. The grammar is still the grammar; the preprocessor is a thin layer wrapping it. When the boundary between what you can change and what you can't is drawn honestly, each extension stays small, and the combined system stays buildable.

Appendix A. File Manifest

New and modified source files introduced by the work described in this paper. Line counts are as of the final build reported in memory.

File	LOC	Role
`aos/claude/ClaudeEventHandler.java`	—	IDE event interface.
`aos/claude/ClaudeApiClient.java`	703	HTTP client, SYSTEM_PROMPT, request builder, auto-fix retry.
`aos/claude/ClaudeTerminalPanel.java`	—	JPanel chat UI (JTextArea + JTextField + Send).
`aos/claude/ClaudeTerminalFrame.java`	—	JInternalFrame wrapper, dockable in Desktop.
`aos/claude/ClaudeLauncher.java`	—	Singleton; openTerminal(), notifyFileCreated(), ...
`aos/jalgenerics/JalGenericsPreprocessor.java`	631	Span classifier, strip(), restore(), erase, helpers.
`aos/jalgenerics/JalGenericsHook.java`	297	Glue around JackCompile.main — preprocess + postprocess.
`aos/jalgenerics/PlanTemplateExpander.java`	472	Template discovery, iterative specialisation, argv rewrite.
`aos/other/MainImproved.java`	Δ 12 lines	Skip .java from JAL input; wrap JackCompile.main with the hook.
`aos/jack/ed/orrrdICl.java`	Δ 4 lines	Claude Terminal menu item; notifyFileCreated from initChild().
`aos/jack/ed/ouucssuo.java`	Δ 3 lines	notifyCompilationStarted / notifyCompilationComplete around compile.
`build.sh / build.bat`	—	Three-step build (compile + restore originals + integration recompile).