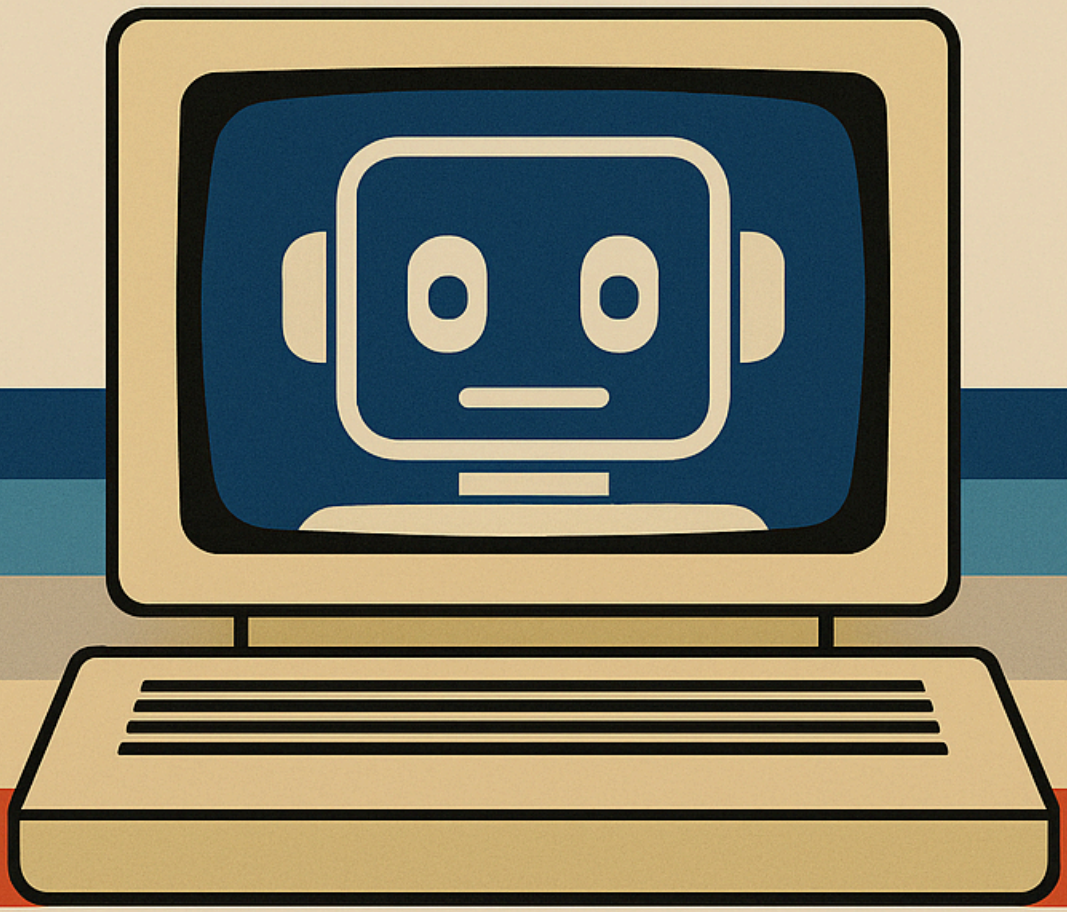
ORION

Project ORION (Mac/OpenAI Edition)

THE AI AGENT USER-DEVELOPER'S GUIDE

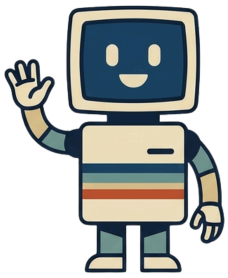


A beginner-friendly tutorial on how to build real, production-grade agent architectures from first principles

Project ORION (Mac/OpenAI Edition)

The AI Agent User-Developer's Guide

A beginner-friendly tutorial on how to build real, production-grade agent architectures from first principles.



Welcome to the future of computing!

Table of Contents

ORION = **O**pen-source **R**eplicant of an **I**ntelligent **O**perati**N**g agent

Introduction

Congratulations! You're about to build one of the most exciting things in modern computing: a real AI agent that can actually do things (create spreadsheets, make presentations, write documents) all by itself, checking its own work along the way.

We know what you might be thinking: "This sounds complicated." And honestly? Under the hood, it is. But so is every powerful tool. What matters is that we're going to take this step by step. By the end you'll not only have a working system, you'll also understand how every piece fits together.

This guide is designed with a simple philosophy: we don't skip steps, we don't assume you already know things, and we celebrate when stuff works. If you've ever felt frustrated by tutorials that leave out the "obvious" parts (which aren't obvious at all if you're just learning), this guide is for you.

Here's what you're going to build:

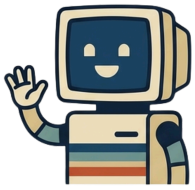
- An AI Agent that receives tasks in plain English
- Tools that create real documents—spreadsheets, presentations, word docs
- A quality control system that looks at what it created and checks if it's right
- A feedback loop that fixes mistakes automatically

And here's something that might surprise you:

> The architecture you're about to build isn't a simplified tutorial version or a "learning exercise." This is the real thing—the same design patterns used in production-grade AI systems at major labs. The agent loop, the Protocol Buffer intermediate representation, the render-and-verify QA pipeline with vision models, the skill-based routing... this mirrors exactly how the professionals build autonomous agents that ship to millions of users.

You're not learning a toy version. You're building the real architecture, from first principles, on your own machine.

By the time you're done, you'll understand these systems almost as well as the engineers who designed them.



Pretty neat, right?

Let's get started.

Part 1: The Big Picture

Chapter 1: What Are We Actually Building?

The Goal in Plain English

Imagine you could tell your computer:

"Make me a spreadsheet showing quarterly sales with a chart."

And it would:

1. Understand what you want
2. Write the code to create that spreadsheet
3. Actually run that code
4. Look at the result to check if it's correct
5. If something's wrong, fix it and try again
6. Give you the finished file

That's what ORION does. It's a system that creates documents by itself, checking its own work along the way.

Why Is This Hard?

You might think: "Can't ChatGPT already do this?"

Sort of. The language model by itself can write code that creates a spreadsheet. But it can't:

- Actually run that code
- See the result
- Check if the result looks right
- Fix problems and try again

To do all of these things, the model needs a framework that supports it. That's where ORION comes in. It provides the tools and structure the model needs to actually perform useful tasks.

Think of it like this: there's a difference between someone who can write a recipe (model) and someone who can actually cook the meal, taste it, adjust the seasoning, and serve you something delicious (agent).

The Secret Sauce: Render and Verify

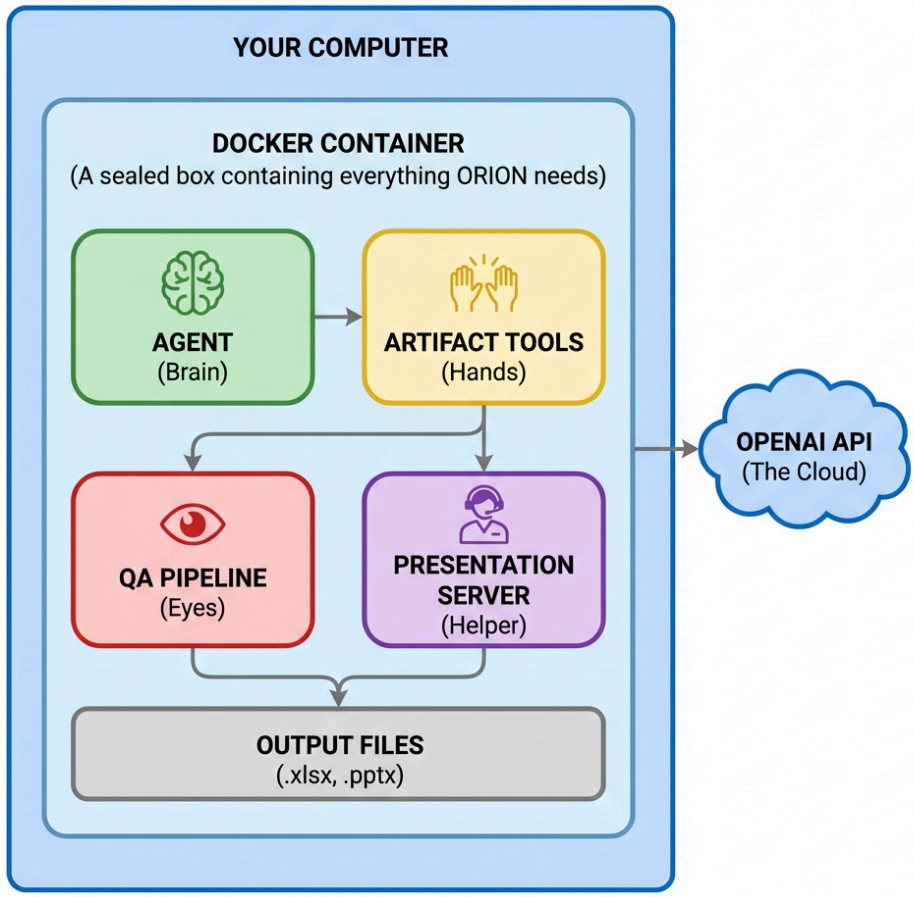
Here's the clever trick that makes ORION work:

```
None
1. The AI writes code to create a document
                        ↓
2. The code runs and creates the document
                        ↓
3. The document gets converted to an image
                        ↓
4. A different AI *looks at the image* and checks if it's correct
                        ↓
5. If not, the original AI gets feedback and tries again
```

This "render-and-verify" loop is the magic. It's like having a chef (the code-writing AI) and a food critic (the vision AI) working together. The critic keeps sending dishes back until they're perfect.

Chapter 2: The Players

ORION has several distinct parts that work together. Don't worry about understanding everything right now—we'll explore each piece in detail. For now, just get the lay of the land.



Let's understand each piece:

The Agent (Brain): This is the orchestrator. It receives your task, decides what to do, asks the AI to write code, runs that code, and coordinates the whole process.

Artifact Tools (Hands): These are the libraries that actually create documents. When the Agent says "make a spreadsheet with these values," the Artifact Tools do the actual work of creating the file.

QA Pipeline (Eyes): This converts documents to images and uses AI vision to check if they look correct. It's the quality control department.

Presentation Server (Helper): PowerPoint files are complicated. This helper service exists just to create them. (We'll explain why later.)

Docker Container: A sealed box that contains everything the system needs to run. This keeps things isolated and reproducible.

OpenAI API: The external AI service that provides the intelligence—both for writing code and for checking images.

Chapter 3: Why These Technologies?

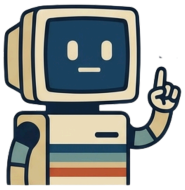
Every technology choice in ORION exists for a reason. Let's understand why each one was chosen.

Why Docker?

What it is: Docker is a way to package software so it runs the same everywhere. Think of it as a shipping container for code—standardized, sealed, and portable.

Why we need it: ORION requires many different pieces of software working together. Installing all of these correctly on your computer is tedious and error-prone. Docker solves this by putting everything in a container that's already configured.

The safety angle: ORION runs AI-generated code. That's potentially dangerous—what if the AI writes something harmful? Docker provides isolation. Code running inside Docker can't easily affect your actual computer. It's like running experiments in a sealed lab.



TIP: Think of Docker as a "computer within your computer." The container has its own operating system, its own installed software, its own file system. When you're done, you can throw it away and start fresh.

Why Protocol Buffers?

What they are: Protocol Buffers (protobufs) are a way to define data structures. They're like a contract that says "a Spreadsheet has Sheets, and each Sheet has Cells, and each Cell has a value and a style."

Why we need them: When building complex systems, you need a clear, precise way to describe your data. Protobufs give you:

- **Precision:** Every field has a specific type. No ambiguity.
- **Validation:** If you try to put the wrong type of data somewhere, you get an error.
- **Documentation:** The .proto files serve as documentation—anyone can read them and understand the data model.

The alternative we avoided: We could use plain dictionaries or JSON. But then there's no guarantee of structure. Someone might forget a field, use the wrong type, or misspell a key. Protobufs catch these errors early.

Why Python?

Python is readable, has extensive libraries for AI and document manipulation, and is what OpenAI's official library is written in. Most AI tools are Python-first.

Why JavaScript for Presentations?

The best PowerPoint generation library (pptxgenjs) is written in JavaScript. Rather than use a worse Python library, we use the right tool for the job. The Python code talks to a small JavaScript server when it needs to create presentations.

This is a common pattern in real systems: use different languages for different tasks, connected by clean interfaces.

Why Vision AI for Quality Assurance?

The problem: How do you know if a generated document is correct?

You could check the data programmatically—verify that cell A1 contains "Revenue"—but this misses visual issues. What if the text is too small to read? What if columns overlap?

The solution: Convert the document to an image and have an AI look at it. Vision models can see formatting issues, layout problems, and visual errors that pure data validation would miss.

This is the same way humans check documents—by looking at them.

Part 2: Setting Up Your Workspace

Chapter 4: Meeting Your Terminal

The terminal (also called command line or shell) is a text-based way to control your computer. Instead of clicking icons, you type commands.

When you see instructions like:

```shell
mkdir orion-agent
```

The text in `green` is a command to type in your terminal, then press RETURN.

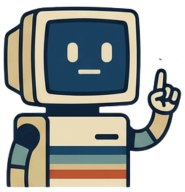
Opening Terminal on Mac

1. Press Command + Space to open Spotlight
2. Type "Terminal"
3. Press Enter

You'll see a window with text, ending in something like:

```none
yourname@your-mac ~ %
```

This is the "prompt"—the computer is waiting for you to type something.



TIP: The terminal might look intimidating at first, but remember: you can't break anything by typing commands. If something goes wrong, you can always close the window and start fresh. The best way to learn is to try things and see what happens.

Basic Commands You'll Need

Let's try some commands. Type each one and press RETURN to see what happens.

pwd – "Print Working Directory"

Shows where you currently are in the file system.

```
Shell
pwd
```

You should see something like: `/Users/yourname`

ls – "List"

Shows files and folders in the current location.

```
Shell
ls
```

You should see folders like: `Desktop Documents Downloads`

cd – "Change Directory"

Moves you to a different folder.

```
Shell
cd Documents
```

Now type `pwd` again—you should be in `/Users/yourname/Documents`

To go back up one level:

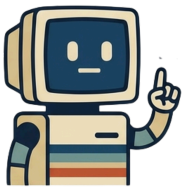
```shell
cd ..
```

mkdir – "Make Directory"

Creates a new folder.

```shell
mkdir test-folder
```

Now type `ls`—you should see your new folder!



TIP: Made a mistake? Press the up arrow key to see your previous commands. You can edit them and try again.

What Those Symbols Mean

You'll see these symbols throughout this guide:

- `~` – Your home directory (e.g., `/Users/yourname`)
- `/` – The root of the file system, or a separator between folder names
- `.` – The current directory
- `..` – The parent directory (one level up)

Chapter 5: Installing Homebrew

What Is Homebrew?

Homebrew is a "package manager" for Mac. It's a tool that installs other tools.

Without Homebrew, installing software means going to websites, downloading installers, running them, and maybe configuring things manually.

With Homebrew:

```shell
brew install python
```

One command. Done.

Installing Homebrew

Open Terminal and paste this command:

```shell
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.s
h)"
```

Press RETURN. The script will:

1. Explain what it's going to do
2. Ask for your password (for installing things system-wide)
3. Download and install Homebrew

This takes a few minutes. Be patient—a lot is happening behind the scenes.

After installation, you might need to add Homebrew to your PATH. The script will tell you if so—just follow the instructions it displays.

Verifying Homebrew Works

```shell
brew --version
```

You should see something like: `Homebrew 4.x.x`

If you see that, you're ready to move on!

Chapter 6: Installing Docker

What Is Docker, Really?

Imagine you could take a snapshot of an entire computer—operating system, all installed software, all configurations—and package it into a single file. Then anyone could run that "computer" inside their own computer.

That's essentially what Docker does.

A **Docker image** is like a snapshot—a template for a virtual computer.

A **Docker container** is a running instance of that image—an actual virtual computer executing code.

Installing Docker Desktop

```shell
brew install --cask docker
```

What's `--cask`? Homebrew distinguishes between regular packages (command-line tools) and casks (full applications with graphical interfaces). Docker Desktop is a full application, so it's a cask.

Starting Docker

After installation:

1. Open Docker Desktop from your Applications folder
2. Wait for it to start (you'll see a whale icon in your menu bar)
3. The whale icon should stop animating when Docker is ready

Verifying Docker Works

Let's try it out:

```shell
docker --version
```

You should see: `Docker version 24.x.x` or similar.

Now let's run a test container:

```shell
docker run hello-world
```

Docker will download a test image and print a welcome message. If you see "Hello from Docker!" then everything is working!

Chapter 7: Installing the Rest

Let's install the remaining tools we need. You can do this all in one command:

```shell
brew install --cask visual-studio-code
brew install --cask libreoffice
brew install git protobuf python@3.11 node@22 poppler graphviz
```

Let's understand what each of these is:

- **Visual Studio Code:** A code editor with syntax highlighting, auto-completion, and an integrated terminal.
- **LibreOffice:** A free office suite. We need it to convert documents to PDF for the QA pipeline.
- **Git:** Version control—tracks changes to your code over time.
- **protobuf:** The Protocol Buffer compiler. It reads .proto files and generates Python code.
- **python@3.11:** Python version 3.11.
- **node@22:** Node.js version 22, for running JavaScript outside a web browser.
- **poppler:** PDF utilities, including pdftoppm which converts PDF pages to images.
- **graphviz:** A diagramming tool used by some Python libraries.

Verifying Everything Works

Try these commands:

```shell
code --version     # Visual Studio Code
git --version      # Git
protoc --version   # Protocol Buffers
python3 --version  # Python
```

```
node --version     # Node.js
```

Each should display a version number. If any command shows "not found," re-run the installation for that tool.

Part 3: Building the Project

Chapter 8: Creating Your Project Folder

Creating the Directory Structure

Let's create the folder structure for ORION:

```shell
cd ~
mkdir orion-agent
cd orion-agent
git init
```

Now create all the subdirectories:

```shell
mkdir -p src/agent
mkdir -p src/artifact_tool
mkdir -p src/proto
mkdir -p src/generated
mkdir -p src/qa_pipeline
mkdir -p src/skills/spreadsheet
mkdir -p src/skills/presentation
mkdir -p src/skills/document
mkdir -p js_pipeline
mkdir -p tests
```

```
mkdir -p output
```

What Each Directory Is For

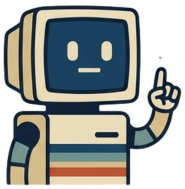
```
None
orion-agent/
├── src/                    # All Python source code
│   ├── agent/              # The main orchestrator (brain)
│   ├── artifact_tool/      # Tools for creating documents
(hands)
│   ├── proto/              # Protocol Buffer definitions
(blueprints)
│   ├── generated/          # Code generated from protobufs
│   ├── qa_pipeline/        # Quality assurance code (eyes)
│   └── skills/             # Instructions for different document
types
│       ├── spreadsheet/    # Spreadsheet skill guide
│       ├── presentation/   # Presentation skill guide
│       └── document/       # Document skill guide
├── js_pipeline/            # JavaScript code for presentations
├── tests/                  # Automated tests
└── output/                 # Where generated files go
```

Creating Python Package Files

Python needs special files to recognize folders as code packages. Let's create them:

```
Shell
touch src/__init__.py
touch src/agent/__init__.py
touch src/artifact_tool/__init__.py
touch src/generated/__init__.py
touch src/qa_pipeline/__init__.py
touch tests/__init__.py
```

These files can be empty—their mere existence tells Python "this is a package."



TIP: The `touch` command creates an empty file if it doesn't exist, or updates its timestamp if it does. It's a quick way to create placeholder files.

Let's verify the structure:

```shell
ls -la
```

You should see all your folders plus the hidden `.git` directory.

Chapter 9: Creating Configuration Files

Before we write any code, we need a few configuration files.

The .gitignore File

This tells Git which files to ignore (temporary files, secrets, generated code):

```shell
cat > .gitignore << 'EOF'
# Python
__pycache__/
*.py[cod]
*.so
venv/

# IDE
.vscode/
.idea/
*.swp
```

```
# macOS
.DS_Store

# Secrets - NEVER commit these
.env

# Node
node_modules/

# Output files
output/
*.tmp
*.log
EOF
```

What's happening here? The `cat > filename << 'EOF'` pattern lets you create a file with multiple lines. Everything between `<< 'EOF'` and `EOF` becomes the file's content.

The .dockerignore File

This tells Docker which files to skip when building images, making builds faster:

Shell

```
cat > .dockerignore << 'EOF'
# Version control
.git
.gitignore

# Secrets
.env

# Output and temp files
output/
*.tmp
*.log
```

```
# Python
__pycache__/
*.py[cod]
venv/

# Node (we reinstall in container)
node_modules/

# IDE
.vscode/
.idea/

# macOS
.DS_Store

# Documentation
*.md
!src/skills/**/*.md
EOF
```

Python Dependencies (requirements.txt)

This lists all the Python libraries ORION needs:

Shell

```
cat > requirements.txt << 'EOF'
# Core dependencies
openai==1.54.0
protobuf==5.28.3

# Spreadsheet support
openpyxl==3.1.5
Pycel==1.0b30
```

```
# Document support
python-docx==1.1.2
reportlab==4.2.5

# Image processing
pdf2image==1.17.0
Pillow==11.0.0

# HTTP client
requests==2.32.3

# Testing
pytest==8.3.4
pytest-cov==6.0.0

# Security (for production sandboxing)
RestrictedPython==7.4
EOF
```

Why specify versions? Code evolves. Version 1.0 of a library might work differently than version 2.0. By specifying `openpyxl==3.1.5`, we ensure everyone gets identical results.

JavaScript Dependencies (package.json)

Create the JavaScript configuration:

Shell

```
cat > js_pipeline/package.json << 'EOF'
{
  "name": "orion-presentation-pipeline",
  "version": "1.0.0",
  "description": "Presentation generation service for Project
ORION",
  "main": "server.js",
  "scripts": {
```

```
    "start": "node server.js"
  },
  "dependencies": {
    "express": "4.21.2",
    "pptxgenjs": "3.12.0"
  }
}
EOF
```

Part 4: Protocol Buffers: Your Data Blueprints

Chapter 10: What Problem Do Protobufs Solve?

Imagine you're building a system where Python code creates data, that data gets sent to JavaScript code, which creates a file, which gets sent back to Python for verification.

How do you ensure everyone agrees on what the data looks like?

Option 1: Just use dictionaries

python

```Python
spreadsheet = {
    "sheets": [
        {
            "name": "Sheet1",
            "cells": {
                "A1": {"value": "Hello"}
            }
        }
    ]
}
```

The problems:

- No validation. What if someone types "cel" instead of "cells"?

- No documentation. What fields are valid?
- No consistency across different parts of the codebase.

Option 2: Use Protocol Buffers

You define the structure formally:

protobuf

```protobuf
message Spreadsheet {
    repeated Sheet sheets = 1;
}

message Sheet {
    string name = 1;
    map<string, Cell> cells = 2;
}
```

Now:

- The structure is documented and precise
- Code is generated that enforces the structure
- Using a wrong field name gives an error
- The same definition works across languages

The Compilation Process

```
.proto files ──▸ protoc compiler ──▸ Python classes
                                 ──▸ JavaScript classes
                                 ──▸ Any other language
```

You write the definition once. The compiler generates code for whatever languages you need.

Chapter 11: Anatomy of a .proto File

Let's create our first Protocol Buffer file and understand every line.

Shell

```
cat > src/proto/common.proto << 'EOF'
syntax = "proto3";

package orion.artifact;

message Color {
    string rgb = 1;
}

message TextStyle {
    optional string font_face = 1;
    optional float font_size = 2;
    optional bool bold = 3;
    optional bool italic = 4;
    optional bool underline = 5;
    optional Color color = 6;
}

message CellStyle {
    optional TextStyle text_style = 1;
    optional string horizontal_align = 2;
    optional string vertical_align = 3;
    optional bool wrap_text = 4;
}

message Fill {
    oneof fill_type {
        Color solid_fill = 1;
        GradientFill gradient_fill = 2;
    }
}

message GradientFill {
    repeated GradientStop stops = 1;
}
```

```
message GradientStop {
    float position = 1;
    Color color = 2;
}
EOF
```

Let's break this down line by line:

`syntax = "proto3";`

This declares we're using Protocol Buffers version 3 (the latest major version).

`package orion.artifact;`

A namespace to avoid naming conflicts. If another project has a "Color" message, they won't collide because ours is `orion.artifact.Color`.

`message Color { ... }`

Defines a data structure (like a class in object-oriented programming).

`string rgb = 1;`

A field named "rgb" of type "string". The `= 1` is the field number (used internally for efficiency), not a default value.

`optional`

Means the field might not be present. Without `optional`, you can't distinguish "explicitly set to empty" from "never set."

`oneof`

Means "exactly one of these." A Fill is either a solid color OR a gradient, never both.

`repeated`

A list of things. `repeated GradientStop stops` means there can be zero, one, or many stops.

Chapter 12: Creating the Document Blueprints

Now let's create the blueprints for each document type.

Spreadsheet Blueprint

```shell
cat > src/proto/spreadsheet.proto << 'EOF'
syntax = "proto3";

import "proto/common.proto";

package orion.artifact;

message Spreadsheet {
    string artifact_id = 1;
    repeated Sheet sheets = 2;
}

message Sheet {
    string sheet_id = 1;
    string name = 2;
    map<string, Cell> cells = 3;
}

message Cell {
    optional string string_value = 1;
    optional double number_value = 2;
    optional bool bool_value = 3;
    optional string formula = 4;
    optional CellStyle style = 5;
    optional Fill fill = 6;
}
EOF
```

Notice `map<string, Cell> cells`. This is a dictionary where keys are cell addresses like "A1" and values are Cell objects.

Presentation Blueprint

Shell

```
cat > src/proto/presentation.proto << 'EOF'
syntax = "proto3";

import "proto/common.proto";

package orion.artifact;

message Presentation {
    string artifact_id = 1;
    repeated Slide slides = 2;
    optional float slide_width = 3;
    optional float slide_height = 4;
}

message Slide {
    string slide_id = 1;
    repeated SlideObject objects = 2;
}

message SlideObject {
    float x = 1;
    float y = 2;
    float w = 3;
    float h = 4;
    oneof object_type {
        TextBox text_box = 5;
        Image image = 6;
        Shape shape = 7;
    }
}

message TextBox {
    string text = 1;
    optional TextStyle style = 2;
    optional string align = 3;
    optional string valign = 4;
```

```
}

message Image {
    bytes image_data = 1;
    string content_type = 2;
}

message Shape {
    string shape_type = 1;
    optional Fill fill = 2;
    optional Color border_color = 3;
    optional float border_width = 4;
}
EOF
```

Document Blueprint

Shell

```
cat > src/proto/document.proto << 'EOF'
syntax = "proto3";

import "proto/common.proto";

package orion.artifact;

message Document {
    string artifact_id = 1;
    repeated Paragraph paragraphs = 2;
}

message Paragraph {
    repeated TextRun runs = 1;
    optional string style = 2;
}
```

```
message TextRun {
    string text = 1;
    optional TextStyle style = 2;
}
EOF
```

Chapter 13: Compiling the Blueprints

Now we turn our .proto files into Python code:

```shell
protoc -I=src --python_out=src/generated src/proto/*.proto
```

Let's break this down:

- `protoc`: The Protocol Buffer compiler
- `-I=src`: "Include path"—where to find imports (so `import "proto/common.proto"` resolves to `src/proto/common.proto`)
- `--python_out=src/generated`: Generate Python code in this directory
- `src/proto/*.proto`: Compile all .proto files

After running this, check what was created:

```shell
ls -la src/generated/proto/
```

You should see files like:

- `common_pb2.py`
- `spreadsheet_pb2.py`
- `presentation_pb2.py`
- `document_pb2.py`

Create the package init file:

```shell
touch src/generated/proto/__init__.py
```

Why `_pb2`? The "2" is historical—it distinguished proto2 from proto1. Even with proto3, the convention stuck.

Congratulations! You've just created your first Protocol Buffer schema and compiled it to Python code. These generated files contain classes that enforce your data structure automatically.

Part 5: The Artifact System: Building Documents

Chapter 14: What Is an Artifact?

In ORION, an "artifact" is a document we're creating—a spreadsheet, presentation, or Word document.

Each artifact type has:

1. A Protocol Buffer definition (the data model)
2. A Python class (the interface for manipulating it)
3. An export method (to save as a real file)

Let's build these classes.

The Base Class

First, we create a base class that all artifacts inherit from:

```shell
cat > src/artifact_tool/base.py << 'EOF'
"""Base class for all artifact types."""

from abc import ABC, abstractmethod


class BaseArtifact(ABC):
```

```
    """Abstract base class for artifacts."""

    def __init__(self, artifact_id: str):
        self.artifact_id = artifact_id
        self._proto = None

    @abstractmethod
    def export(self, path: str):
        """Export the artifact to its native format."""
        pass
EOF
```

What's `ABC`? "Abstract Base Class." A class that can't be used directly—it's a template for other classes.

What's `@abstractmethod`? A decorator marking a method that subclasses MUST implement. If you create a class that inherits from `BaseArtifact` without implementing `export()`, Python raises an error.

Chapter 15: The Spreadsheet Artifact

Now let's build the spreadsheet artifact. This is the most complex one, so we'll go through it carefully.

Shell

```
cat > src/artifact_tool/spreadsheet.py << 'EOF'
"""Spreadsheet artifact implementation."""

import openpyxl
from openpyxl.styles import Font, Alignment, PatternFill
from src.generated.proto.spreadsheet_pb2 import Spreadsheet,
Sheet, Cell
from src.generated.proto.common_pb2 import CellStyle, TextStyle,
Color, Fill
from .base import BaseArtifact
```

```
class SpreadsheetArtifact(BaseArtifact):
    """Spreadsheet artifact using Protocol Buffers."""

    def __init__(self, artifact_id: str):
        super().__init__(artifact_id)
        self._proto = Spreadsheet(artifact_id=artifact_id)

    def get_sheet(self, name: str):
        """Get or create a sheet by name."""
        # Look for existing sheet
        for sheet_proto in self._proto.sheets:
            if sheet_proto.name == name:
                return SheetHandle(sheet_proto)

        # Create new sheet if not found
        new_sheet = self._proto.sheets.add()
        new_sheet.name = name
        new_sheet.sheet_id = f"sheet_{len(self._proto.sheets)}"
        return SheetHandle(new_sheet)

    def recalculate(self):
        """Recalculate all formulas."""
        from .formula_engine import FormulaEngine
        engine = FormulaEngine(self._proto)
        engine.recalculate()

    def export_to_xlsx(self, path: str):
        """Export to Excel format."""
        wb = openpyxl.Workbook()

        # Handle the case where no sheets have been defined
        if not self._proto.sheets:
            # Just rename the default sheet and save
            wb.active.title = "Sheet1"
```

```
        wb.save(path)
        print(f"Exported empty spreadsheet to {path}")
        return

    # Remove default empty sheet since we have sheets to add
    wb.remove(wb.active)

    for sheet_proto in self._proto.sheets:
        ws = wb.create_sheet(title=sheet_proto.name)

        for address, cell_proto in sheet_proto.cells.items():
            cell = ws[address]

            # Set value based on type
            if cell_proto.HasField("formula"):
                cell.value = cell_proto.formula
            elif cell_proto.HasField("string_value"):
                cell.value = cell_proto.string_value
            elif cell_proto.HasField("number_value"):
                cell.value = cell_proto.number_value
            elif cell_proto.HasField("bool_value"):
                cell.value = cell_proto.bool_value

            # Apply styling if present
            if cell_proto.HasField("style"):
                style = cell_proto.style
                if style.HasField("text_style"):
                    ts = style.text_style
                    font_kwargs = {}
                    if ts.HasField("font_face"):
                        font_kwargs["name"] = ts.font_face
                    if ts.HasField("font_size"):
                        font_kwargs["size"] = ts.font_size
                    if ts.HasField("bold"):
                        font_kwargs["bold"] = ts.bold
                    if ts.HasField("italic"):
```

```
                            font_kwargs["italic"] = ts.italic
                        if font_kwargs:
                            cell.font = Font(**font_kwargs)

                    if style.HasField("horizontal_align"):
                        cell.alignment =
Alignment(horizontal=style.horizontal_align)

                # Apply fill if present
                if cell_proto.HasField("fill"):
                    fill = cell_proto.fill
                    if fill.HasField("solid_fill"):
                        cell.fill = PatternFill(
                            start_color=fill.solid_fill.rgb,
                            end_color=fill.solid_fill.rgb,
                            fill_type="solid"
                        )

        wb.save(path)
        print(f"Exported spreadsheet to {path}")

    def export(self, path: str):
        """Export artifact (delegates to export_to_xlsx)."""
        self.export_to_xlsx(path)


class SheetHandle:
    """Handle for manipulating a sheet."""

    def __init__(self, sheet_proto: Sheet):
        self._sheet_proto = sheet_proto

    def set_cell(self, address: str, value=None, formula=None,
style=None):
        """Set a cell's value, formula, or style."""
```

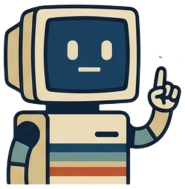
```
        # Protobuf maps auto-create entries on access, so we just
access directly
        cell = self._sheet_proto.cells[address]

        if formula is not None:
            cell.formula = formula
        elif value is not None:
            # Check bool BEFORE int (bool is a subclass of int in
Python!)
            if isinstance(value, bool):
                cell.bool_value = value
            elif isinstance(value, str):
                cell.string_value = value
            elif isinstance(value, (int, float)):
                cell.number_value = float(value)

        if style is not None:
            cell.style.CopyFrom(style)

    def __getitem__(self, address: str):
        """Allow sheet["A1"] syntax."""
        return self._sheet_proto.cells.get(address)

    def __setitem__(self, address: str, value):
        """Allow sheet["A1"] = value syntax."""
        self.set_cell(address, value=value)
EOF
```



TIP: Notice `__getitem__` and `__setitem__`—these are Python "magic methods" that let you use `sheet["A1"]` syntax instead of `sheet.get_cell("A1")`. Little touches like this make code more intuitive.

Chapter 16: The Formula Engine

When you write `=SUM(A1:A3)` in Excel, Excel calculates the result. We need to do the same thing.

Pycel is a library that evaluates Excel formulas. But it has a quirk: it needs an actual Excel file to work with.

Our solution: create a temporary file, let Pycel evaluate it, then clean up.

Shell

```
cat > src/artifact_tool/formula_engine.py << 'EOF'
"""Formula evaluation engine for spreadsheets."""

import openpyxl
import tempfile
import os
from pycel import ExcelCompiler


class FormulaEngine:
    """Evaluates Excel formulas in spreadsheet artifacts."""

    def __init__(self, artifact_proto):
        self._artifact_proto = artifact_proto

    def recalculate(self):
        """Recalculate all formulas in the spreadsheet."""

        # Convert our protobuf to an openpyxl workbook
        wb = self._proto_to_openpyxl()

        # Save to a temporary file (pycel requires a file path)
        with tempfile.NamedTemporaryFile(delete=False,
suffix=".xlsx") as tmp:
            wb.save(tmp.name)
            tmp_path = tmp.name
```

```
        try:
            # Load the temp file with Pycel
            excel = ExcelCompiler(filename=tmp_path)

            # For each formula cell, evaluate it
            for sheet_proto in self._artifact_proto.sheets:
                for address, cell_proto in
sheet_proto.cells.items():
                    if cell_proto.HasField("formula"):
                        try:
                            cell_ref =
f"{sheet_proto.name}!{address}"
                            value = excel.evaluate(cell_ref)

                            # Store the calculated value
                            if isinstance(value, (int, float)):
                                cell_proto.number_value =
float(value)
                            elif isinstance(value, str):
                                cell_proto.string_value = value
                            elif isinstance(value, bool):
                                cell_proto.bool_value = value
                        except Exception as e:
                            print(f"Error evaluating formula in
{address}: {e}")
                            cell_proto.string_value = "#ERROR!"
        finally:
            # Always clean up the temp file
            os.remove(tmp_path)

    def _proto_to_openpyxl(self):
        """Convert protobuf to openpyxl workbook."""
        wb = openpyxl.Workbook()
        wb.remove(wb.active)

        for sheet_proto in self._artifact_proto.sheets:
```

```
            ws = wb.create_sheet(title=sheet_proto.name)

            for address, cell_proto in sheet_proto.cells.items():
                if cell_proto.HasField("formula"):
                    ws[address].value = cell_proto.formula
                elif cell_proto.HasField("string_value"):
                    ws[address].value = cell_proto.string_value
                elif cell_proto.HasField("number_value"):
                    ws[address].value = cell_proto.number_value
                elif cell_proto.HasField("bool_value"):
                    ws[address].value = cell_proto.bool_value

        return wb
EOF
```

Why `finally`? The `finally` block runs whether the code succeeds or fails. This ensures we always clean up the temporary file, even if an error occurs.

Chapter 17: The Presentation and Document Artifacts

Let's create the remaining artifacts. These follow the same pattern as the spreadsheet.

Presentation Artifact

Shell

```
cat > src/artifact_tool/presentation.py << 'EOF'
"""Presentation artifact implementation."""

import os
import requests
import shutil
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
from google.protobuf.json_format import MessageToDict
from src.generated.proto.presentation_pb2 import Presentation,
Slide, SlideObject, TextBox, Image, Shape
```

```
from src.generated.proto.common_pb2 import TextStyle, Color, Fill
from .base import BaseArtifact


class PresentationArtifact(BaseArtifact):
    """Presentation artifact using Protocol Buffers."""

    def __init__(self, artifact_id: str):
        super().__init__(artifact_id)
        self._proto = Presentation(artifact_id=artifact_id)
        self._proto.slide_width = 10.0
        self._proto.slide_height = 7.5

    def add_slide(self):
        """Add a new slide and return a handle."""
        slide = self._proto.slides.add()
        slide.slide_id = f"slide_{len(self._proto.slides)}"
        return SlideHandle(slide)

    def export_to_pptx(self, path: str):
        """Export via the JavaScript pipeline."""
        # Convert protobuf to dictionary
        artifact_dict = MessageToDict(self._proto,
preserving_proto_field_name=True)

        # Set up retry logic for robustness
        session = requests.Session()
        retries = Retry(total=3, backoff_factor=1,
status_forcelist=[500, 502, 503, 504])
        session.mount('http://',
HTTPAdapter(max_retries=retries))

        # Get server URL from environment or use default
        server_url = os.environ.get("PRESENTATION_SERVER_URL",
"http://localhost:3000")
```

```
        try:
            response = session.post(
                f"{server_url}/create-presentation",
                headers={"Content-Type": "application/json"},
                json=artifact_dict,
                timeout=30
            )
            response.raise_for_status()
            result = response.json()

            # Copy from server path to requested path only if
different
            server_path = result.get("path")
            if server_path and os.path.exists(server_path):
                # Resolve both paths to absolute to check if
they're the same
                abs_server_path = os.path.abspath(server_path)
                abs_dest_path = os.path.abspath(path)

                if abs_server_path != abs_dest_path:
                    shutil.copy(server_path, path)
                print(f"Presentation exported successfully to
{path}")
            else:
                raise RuntimeError(f"Server file not found at
{server_path}")

        except requests.exceptions.ConnectionError:
            raise RuntimeError(
                f"Cannot connect to presentation server at
{server_url}. "
                "Make sure the Node.js server is running (node
js_pipeline/server.js)"
            )
        except requests.exceptions.RequestException as e:
```

```
            raise RuntimeError(f"Presentation pipeline error:
{e}")

    def export(self, path: str):
        self.export_to_pptx(path)


class SlideHandle:
    """Handle for manipulating a slide."""

    def __init__(self, slide_proto: Slide):
        self._slide_proto = slide_proto

    def add_text(self, text: str, x: float, y: float, w: float,
h: float,
                 font_size: float = 18, bold: bool = False,
align: str = "left"):
        """Add a text box to the slide."""
        obj = self._slide_proto.objects.add()
        obj.x = x
        obj.y = y
        obj.w = w
        obj.h = h

        obj.text_box.text = text
        obj.text_box.style.font_size = font_size
        obj.text_box.style.bold = bold
        obj.text_box.align = align
EOF
```

Document Artifact

Shell

```
cat > src/artifact_tool/document.py << 'EOF'
"""Document artifact implementation."""
```

```
from docx import Document as DocxDocument
from docx.shared import Pt
from src.generated.proto.document_pb2 import Document, Paragraph,
TextRun
from src.generated.proto.common_pb2 import TextStyle
from .base import BaseArtifact


class DocumentArtifact(BaseArtifact):
    """Document artifact using Protocol Buffers."""

    def __init__(self, artifact_id: str):
        super().__init__(artifact_id)
        self._proto = Document(artifact_id=artifact_id)

    def add_paragraph(self, text: str, style: str = "Normal",
bold: bool = False):
        """Add a paragraph to the document."""
        para = self._proto.paragraphs.add()
        para.style = style

        run = para.runs.add()
        run.text = text
        if bold:
            run.style.bold = True

    def export_to_docx(self, path: str):
        """Export to Word format."""
        doc = DocxDocument()

        for para_proto in self._proto.paragraphs:
            # Determine style
            style = para_proto.style if para_proto.style else
"Normal"
            para = doc.add_paragraph(style=style)
```

```
            for run_proto in para_proto.runs:
                run = para.add_run(run_proto.text)

                if run_proto.HasField("style"):
                    if run_proto.style.HasField("bold"):
                        run.bold = run_proto.style.bold
                    if run_proto.style.HasField("italic"):
                        run.italic = run_proto.style.italic
                    if run_proto.style.HasField("font_size"):
                        run.font.size =
Pt(run_proto.style.font_size)

        doc.save(path)
        print(f"Exported document to {path}")

    def export(self, path: str):
        self.export_to_docx(path)
EOF
```

Part 6: The QA Pipeline: Checking Your Work

Chapter 18: How Quality Assurance Works

The QA pipeline is what makes ORION special. It doesn't just create documents—it checks them.

Here's the process:

```
None
Artifact
    ↓
Export to native format (.xlsx, .pptx, .docx)
    ↓
```

```
Convert to PDF (using LibreOffice)
    ↓
Convert PDF to PNG images (using pdftoppm)
    ↓
Send images to Vision AI with a checklist
    ↓
Vision AI looks at the images and responds: PASS or FAIL
    ↓
If FAIL, feedback goes back to the Agent to try again
```

Why Images?

You might wonder: why convert to images? Why not just check the data?

Because visual problems can't be caught by data validation. What if:

- Text is too small to read?
- Columns overlap?
- A chart is malformed?
- Colors clash?

These are things you can only catch by looking. So we let an AI look.

Chapter 19: Building the QA Pipeline

Shell

```
cat > src/qa_pipeline/verify.py << 'EOF'
"""Quality assurance pipeline with vision model verification."""

import subprocess
import os
import base64
from openai import OpenAI
import tempfile
```

```
def render_and_verify(artifact, checklist: list[str], timeout:
int = 60):
    """Render artifact to images and verify with vision model.

    Args:
        artifact: The artifact to verify
        checklist: List of criteria to check
        timeout: Timeout in seconds for conversion operations
(default: 60)

    Returns:
        dict with 'passed' (bool) and 'feedback' (str)
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        artifact_type = artifact.__class__.__name__

        # Step 1: Export to native format
        if artifact_type == "SpreadsheetArtifact":
            native_path = os.path.join(tmpdir, "export.xlsx")
            artifact.export_to_xlsx(native_path)
        elif artifact_type == "PresentationArtifact":
            native_path = os.path.join(tmpdir, "export.pptx")
            artifact.export_to_pptx(native_path)
        elif artifact_type == "DocumentArtifact":
            native_path = os.path.join(tmpdir, "export.docx")
            artifact.export_to_docx(native_path)
        else:
            raise TypeError(f"Unsupported artifact type:
{artifact_type}")

        # Step 2: Convert to PDF using LibreOffice
        try:
            subprocess.run([
                "soffice",
                "--headless",
                "--convert-to", "pdf",
```

```
                "--outdir", tmpdir,
                native_path
            ], check=True, timeout=timeout)
        except subprocess.TimeoutExpired:
            return {"passed": False, "feedback": "LibreOffice
conversion timed out. Try increasing the timeout or restarting
the container."}
        except subprocess.CalledProcessError as e:
            return {"passed": False, "feedback": f"LibreOffice
conversion failed: {e}"}
        except FileNotFoundError:
            return {"passed": False, "feedback": "LibreOffice
(soffice) not found. Make sure it's installed."}

        # Step 3: Convert PDF to PNG images
        # Get the base filename without extension for PDF path
construction
        base_filename =
os.path.splitext(os.path.basename(native_path))[0]
        pdf_path = os.path.join(tmpdir, f"{base_filename}.pdf")
        image_prefix = os.path.join(tmpdir, "page")

        try:
            subprocess.run([
                "pdftoppm",
                "-png",
                pdf_path,
                image_prefix
            ], check=True, timeout=timeout)
        except subprocess.TimeoutExpired:
            return {"passed": False, "feedback": "PDF to PNG
conversion timed out."}
        except FileNotFoundError:
            return {"passed": False, "feedback": "pdftoppm not
found. Make sure poppler-utils is installed."}
        except Exception as e:
```

```
            return {"passed": False, "feedback": f"PDF to PNG
conversion failed: {e}"}

        # Step 4: Collect all PNG images
        image_paths = sorted([
            os.path.join(tmpdir, f)
            for f in os.listdir(tmpdir)
            if f.endswith(".png")
        ])

        if not image_paths:
            return {"passed": False, "feedback": "No images were
generated from the document."}

        # Step 5: Encode images to base64 for the API
        encoded_images = []
        for image_path in image_paths:
            with open(image_path, "rb") as img_file:

encoded_images.append(base64.b64encode(img_file.read()).decode("u
tf-8"))

        # Step 6: Call vision model for verification
        try:
            client = OpenAI()  # Reads OPENAI_API_KEY from
environment

            checklist_text = "\n".join(f"- {item}" for item in
checklist)

            response = client.chat.completions.create(
                model="gpt-4o",
                messages=[
                    {
                        "role": "user",
                        "content": [
```

```
                            {
                                "type": "text",
                                "text": f"""You are a quality
assurance agent. Analyze the provided images and verify against
this checklist:

{checklist_text}

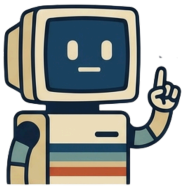
Respond with EXACTLY this format:
PASSED: true or false
FEEDBACK: your detailed notes here"""
                            }
                        ] + [
                            {
                                "type": "image_url",
                                "image_url": {"url":
f"{img}"}
                            } for img in encoded_images
                        ]
                    }
                ],
                max_tokens=1024
            )

            # Parse the response
            content = response.choices[0].message.content
            passed = "PASSED: true" in content or "passed: true"
in content.lower()

            # Extract feedback
            feedback_parts = content.split("FEEDBACK:", 1)
            feedback = feedback_parts[1].strip() if
len(feedback_parts) > 1 else content

            return {"passed": passed, "feedback": feedback}
```

```
        except Exception as e:
            return {"passed": False, "feedback": f"Vision model
error: {e}"}
EOF
```



TIP: The `tempfile.TemporaryDirectory()` context manager creates a temporary folder that automatically gets deleted when we're done. This keeps things clean—no leftover files cluttering up your system.

Part 7: The Agent: The Brain of ORION

Chapter 20: What Is an Agent?

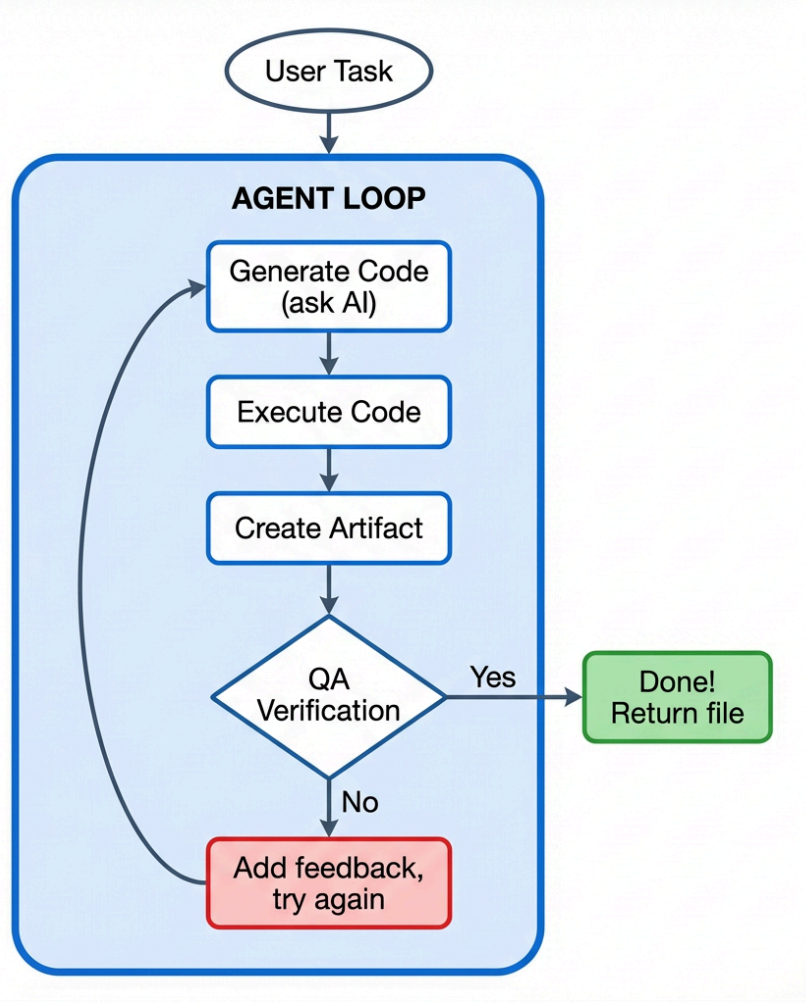
In AI, an "agent" is a program that:

1. Receives a goal
2. Decides on actions to take
3. Takes those actions
4. Observes the results
5. Repeats until the goal is achieved (or gives up)

The ORION agent:

1. Receives a task ("make a spreadsheet with...")
2. Asks the AI to write code
3. Runs that code
4. Checks if the result is correct
5. If not, asks the AI to fix it and try again

The Agent Loop



Chapter 21: Building the Agent

This is the heart of ORION. The agent orchestrates everything: it asks the AI to write code, executes that code, runs quality checks, and iterates until the task is complete.

Two key features make this agent production-ready:

1. **Retry logic with exponential backoff** — If the OpenAI API fails (network issues, rate limits, server errors), the agent waits and tries again instead of crashing.
2. **Dynamic checklist generation** — Instead of using a generic "does it look good?" checklist, the agent asks the AI to generate task-specific quality criteria. A spreadsheet about sales data gets checked for different things than a presentation about company history.

Shell

```
cat > src/agent/main.py << 'EOF'
"""ORION Agent - Main orchestrator."""

import os
import re
import sys
import time
import traceback
from openai import OpenAI
from src.artifact_tool.spreadsheet import SpreadsheetArtifact
from src.artifact_tool.presentation import PresentationArtifact
from src.artifact_tool.document import DocumentArtifact
from src.qa_pipeline.verify import render_and_verify


class OrionAgent:
    """Autonomous agent for document creation."""

    def __init__(self, prompt: str):
        self.prompt = prompt
        self.client = OpenAI()
        self.history = []

        # These are the classes available to the AI-generated
code
        self.local_vars = {
            "SpreadsheetArtifact": SpreadsheetArtifact,
            "PresentationArtifact": PresentationArtifact,
            "DocumentArtifact": DocumentArtifact
        }

        self.max_iterations = 10
        self.skill = self._select_skill(prompt)

    def _select_skill(self, prompt: str):
        """Determine which skill to use based on the prompt."""
```

```
        prompt_lower = prompt.lower()

        # Use word boundaries to avoid false matches (e.g.,
"docker" matching "doc")
        if re.search(r'\b(spreadsheet|excel|xlsx)\b',
prompt_lower):
            return "spreadsheet"
        elif
re.search(r'\b(presentation|slides|powerpoint|pptx)\b',
prompt_lower):
            return "presentation"
        elif re.search(r'\b(document|word|docx)\b',
prompt_lower):
            return "document"
        else:
            return "spreadsheet"  # Default

    def _load_skill_guide(self):
        """Load the skill guide for the current skill."""
        skill_path = f"src/skills/{self.skill}/skill.md"
        if os.path.exists(skill_path):
            with open(skill_path, "r") as f:
                return f.read()
        return ""

    def _manage_context(self):
        """Keep conversation history manageable."""
        # Keep system prompt + last 6 messages
        if len(self.history) > 7:
            self.history = [self.history[0]] + self.history[-6:]

    def _extract_code(self, response_text: str) -> str:
        """Extract Python code from the response."""
        # Look for code blocks (the AI often wraps code in
markdown)
        if "```python" in response_text:
```

```
            code =
response_text.split("```python")[1].split("```")[0]
            return code.strip()
        elif "```" in response_text:
            code = response_text.split("```")[1].split("```")[0]
            return code.strip()
        return response_text.strip()

    def _call_llm_with_retry(self, messages, max_retries=3):
        """Call the LLM with exponential backoff retry logic.

        If the API fails, we wait and try again:
        - 1st retry: wait 1 second
        - 2nd retry: wait 2 seconds
        - 3rd retry: wait 4 seconds

        This handles temporary network issues and rate limits
gracefully.
        """
        for attempt in range(max_retries):
            try:
                response = self.client.chat.completions.create(
                    model="gpt-4-turbo",
                    messages=messages,
                    temperature=0.0,
                    timeout=60
                )
                return response
            except Exception as e:
                if attempt < max_retries - 1:
                    wait_time = 2 ** attempt  # 1s, 2s, 4s
                    print(f"API error (attempt {attempt +
1}/{max_retries}): {e}")
                    print(f"Retrying in {wait_time} seconds...")
                    time.sleep(wait_time)
                else:
```

```
                    print(f"API failed after {max_retries}
attempts: {e}")
                    raise

    def _extract_checklist(self, prompt: str):
        """Generate task-specific quality criteria using AI.

        Instead of a generic checklist, we ask the AI to generate
        criteria specific to this task. A spreadsheet about sales
        gets different checks than a presentation about history.
        """
        try:
            response = self.client.chat.completions.create(
                model="gpt-4-turbo",
                messages=[
                    {
                        "role": "system",
                        "content": "You are a QA expert. Generate
a concise quality checklist."
                    },
                    {
                        "role": "user",
                        "content": f"For this task:
'{prompt}'\n\nGenerate 3-5 specific quality criteria to verify
the output. Return as a simple list, one criterion per line, no
bullet points or numbers."
                    }
                ],
                temperature=0.0,
                max_tokens=200
            )

            # Parse response into list
            content = response.choices[0].message.content
            checklist = [
                line.strip().lstrip("-•*0123456789.").strip()
```

```
                for line in content.split("\n")
                if line.strip()
            ]

            if checklist:
                print(f"Generated QA checklist: {checklist}")
                return checklist
            else:
                return self._default_checklist()

        except Exception as e:
            print(f"Checklist generation failed ({e}), using
defaults")
            return self._default_checklist()

    def _default_checklist(self):
        """Fallback checklist if AI generation fails."""
        return [
            "All requested data is present",
            "Formatting is professional and consistent",
            "No rendering errors or visual artifacts"
        ]

    def run(self):
        """Execute the agent loop."""
        skill_guide = self._load_skill_guide()

        # Generate task-specific QA checklist BEFORE we start
        print("Generating task-specific QA criteria...")
        checklist = self._extract_checklist(self.prompt)

        # Set up the system prompt
        system_prompt = f"""You are an autonomous agent
specialized in creating {self.skill}s.

{skill_guide}
```

```
IMPORTANT RULES:
1. Write ONLY Python code, no explanations
2. Store the completed artifact in a variable called 'artifact'
3. Do NOT include import statements - all classes are already
available
4. Use the artifact classes exactly as shown in the skill
guide"""

        self.history.append({"role": "system", "content":
system_prompt})
        self.history.append({"role": "user", "content":
self.prompt})

        for iteration in range(self.max_iterations):
            print(f"\n{'='*50}")
            print(f"Iteration {iteration +
1}/{self.max_iterations}")
            print(f"{'='*50}")

            # Manage context length
            self._manage_context()

            # Generate code
            print("Generating code...")
            try:
                response =
self._call_llm_with_retry(self.history)
            except Exception:
                print("Failed to get response from LLM.
Aborting.")
                return False

            response_text = response.choices[0].message.content
            code = self._extract_code(response_text)
```

```
            print(f"Generated code:\n{code[:500]}{'...' if
len(code) > 500 else ''}")

            self.history.append({"role": "assistant", "content":
response_text})

            # Execute code
            print("\nExecuting code...")
            try:
                exec(code, {"__builtins__": __builtins__},
self.local_vars)
                exec_success = True
            except Exception as e:
                exec_success = False
                error_msg = traceback.format_exc()
                print(f"Execution error:\n{error_msg}")

            if not exec_success:
                self.history.append({
                    "role": "user",
                    "content": f"That code failed with
error:\n{error_msg}\n\nPlease fix and try again. Remember: no
import statements needed."
                })
                continue

            # Check if artifact was created
            if "artifact" not in self.local_vars:
                self.history.append({
                    "role": "user",
                    "content": "The code ran but you haven't
created the final artifact yet. Create an artifact and store it
in a variable called 'artifact'."
                })
                continue
```

```
            # QA verification
            print("\nRunning QA verification...")
            artifact = self.local_vars["artifact"]
            qa_result = render_and_verify(artifact, checklist)

            print(f"QA Result: {'PASSED' if qa_result['passed']
else 'FAILED'}")
            print(f"Feedback: {qa_result['feedback']}")

            if qa_result["passed"]:
                # Export final artifact
                output_dir = "/app/output" if
os.path.exists("/app/output") else "output"
                os.makedirs(output_dir, exist_ok=True)

                if self.skill == "spreadsheet":
                    output_path =
f"{output_dir}/{artifact.artifact_id}.xlsx"
                    artifact.export_to_xlsx(output_path)
                elif self.skill == "presentation":
                    output_path =
f"{output_dir}/{artifact.artifact_id}.pptx"
                    artifact.export_to_pptx(output_path)
                else:
                    output_path =
f"{output_dir}/{artifact.artifact_id}.docx"
                    artifact.export_to_docx(output_path)

                print(f"\n{'='*50}")
                print(f"SUCCESS! Output saved to: {output_path}")
                print(f"{'='*50}")
                return True
            else:
                self.history.append({
                    "role": "user",
```

```
                    "content": f"The artifact was created but
failed QA verification:\n{qa_result['feedback']}\n\nPlease fix
these issues and regenerate the artifact."
                })

        print("\nMax iterations reached without success.")
        return False


def main():
    """Entry point."""
    # Check for API key
    if "OPENAI_API_KEY" not in os.environ:
        print("Error: OPENAI_API_KEY environment variable not
set")
        print("Set it with: export
OPENAI_API_KEY='your-key-here'")
        sys.exit(1)

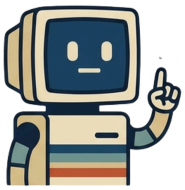
    # Get task from command line, environment variable, or use
default
    if len(sys.argv) > 1:
        task = " ".join(sys.argv[1:])
    elif os.environ.get("ORION_TASK"):
        task = os.environ["ORION_TASK"]
    else:
        task = "Create a spreadsheet with sample quarterly sales
data for 4 products"

    print(f"Starting ORION Agent")
    print(f"Task: {task}\n")

    agent = OrionAgent(task)
    success = agent.run()

    sys.exit(0 if success else 1)
```

```
if __name__ == "__main__":
    main()
EOF
```



TIP: Notice `temperature=0.0` in the API calls. This makes output deterministic—the model always picks the most likely response. For code generation, we want consistency, not creativity.

Part 8: The JavaScript Pipeline

Chapter 22: Why JavaScript for Presentations?

Python is great for many things, but the best PowerPoint generation library (pptxgenjs) is written in JavaScript. Rather than use a worse Python library, we use the right tool for the job.

The architecture is simple:

```
None
Python Code ——HTTP Request——▸ JavaScript Server ——▸ PowerPoint
File
                                        |
                                  Uses pptxgenjs
```

The Python code sends a JSON representation of the presentation. The JavaScript server converts it to a PowerPoint file.

Chapter 23: Building the JavaScript Server

Shell

```
cat > js_pipeline/server.js << 'EOF'
const express = require("express");
const PptxGenJS = require("pptxgenjs");
const fs = require("fs");
const path = require("path");

const app = express();
app.use(express.json({ limit: "50mb" })); // Large limit for
images

// Determine output directory (works both in Docker and locally)
const OUTPUT_DIR = process.env.OUTPUT_DIR ||
    (fs.existsSync("/app/output") ? "/app/output" :
path.join(__dirname, "..", "output"));

// Ensure output directory exists
if (!fs.existsSync(OUTPUT_DIR)) {
    fs.mkdirSync(OUTPUT_DIR, { recursive: true });
}

app.post("/create-presentation", (req, res) => {
    try {
        const artifact = req.body;
        let pres = new PptxGenJS();

        // Set dimensions if specified - define layout BEFORE
setting it
        if (artifact.slide_width && artifact.slide_height) {
            pres.defineLayout({
                name: "CUSTOM",
                width: artifact.slide_width,
                height: artifact.slide_height
            });
            pres.layout = "CUSTOM";
        }
```

```
        // Process each slide
        if (artifact.slides) {
            artifact.slides.forEach(slideData => {
                let slide = pres.addSlide();

                if (slideData.objects) {
                    slideData.objects.forEach(obj => {
                        // Text boxes
                        if (obj.text_box) {
                            const textOpts = {
                                x: obj.x,
                                y: obj.y,
                                w: obj.w,
                                h: obj.h
                            };

                            if (obj.text_box.style) {
                                if (obj.text_box.style.font_size)
{
                                    textOpts.fontSize =
obj.text_box.style.font_size;
                                }
                                if (obj.text_box.style.bold) {
                                    textOpts.bold =
obj.text_box.style.bold;
                                }
                                if (obj.text_box.style.color) {
                                    textOpts.color =
obj.text_box.style.color.rgb;
                                }
                            }

                            if (obj.text_box.align) {
                                textOpts.align =
obj.text_box.align;
                            }
```

```
                            slide.addText(obj.text_box.text,
textOpts);
                        }

                        // Shapes
                        if (obj.shape) {
                            const shapeOpts = {
                                x: obj.x,
                                y: obj.y,
                                w: obj.w,
                                h: obj.h
                            };

                            if (obj.shape.fill &&
obj.shape.fill.solid_fill) {
                                shapeOpts.fill = { color:
obj.shape.fill.solid_fill.rgb };
                            }

                            slide.addShape(pres.shapes.RECTANGLE,
shapeOpts);
                        }
                    });
                }
            });
        }

        // Save the file
        const outputPath = path.join(OUTPUT_DIR,
`${artifact.artifact_id}.pptx`);
        pres.writeFile({ fileName: outputPath })
            .then(() => {
                console.log(`Presentation saved to
${outputPath}`);
```

```
                res.status(200).json({ success: true, path:
outputPath });
            })
            .catch(err => {
                console.error("Error writing presentation:",
err);
                res.status(500).json({ success: false, error:
err.toString() });
            });
    } catch (err) {
        console.error("Error creating presentation:", err);
        res.status(500).json({ success: false, error:
err.toString() });
    }
});

// Health check endpoint
app.get("/health", (req, res) => {
    res.status(200).json({ status: "healthy" });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Presentation pipeline listening on port
${PORT}`);
    console.log(`Output directory: ${OUTPUT_DIR}`);
});
EOF
```

Part 9: Skill Guides: Teaching the AI

Chapter 24: What Are Skill Guides?

Skill guides are instructions we give to the AI about how to use our artifact classes. They're like cheat sheets that get included in the prompt.

Spreadsheet Skill Guide

Shell

```
cat > src/skills/spreadsheet/skill.md << 'EOF'
# Spreadsheet Creation Skill

## Overview
You are creating a spreadsheet using the `SpreadsheetArtifact`
class.

## Basic Usage
```python
# Create a new spreadsheet
artifact = SpreadsheetArtifact("my_spreadsheet")

# Get or create a sheet
sheet = artifact.get_sheet("Sheet1")

# Set cell values
sheet.set_cell("A1", value="Product")
sheet.set_cell("B1", value="Revenue")
sheet.set_cell("A2", value="Widget")
sheet.set_cell("B2", value=50000)

# Set formulas
sheet.set_cell("B10", formula="=SUM(B2:B9)")

# Recalculate formulas
artifact.recalculate()
```

Quality Standards
- Headers should be in row 1
- Use formulas for calculated values, not hardcoded numbers
- Numbers should be numbers, not strings
- Format currency values appropriately
```

```
## Final Step
Store the completed artifact in a variable called `artifact`.
EOF
```

Presentation Skill Guide

Shell

```
cat > src/skills/presentation/skill.md << 'EOF'
# Presentation Creation Skill

## Overview
You are creating a presentation using the `PresentationArtifact`
class.

## Basic Usage
```python
# Create a new presentation
artifact = PresentationArtifact("my_presentation")

# Add a slide
slide = artifact.add_slide()

# Add text to the slide
slide.add_text("Title Text", x=1, y=1, w=8, h=1, font_size=44,
bold=True)
slide.add_text("Body Text", x=1, y=3, w=8, h=3, font_size=24)
```

Quality Standards
- Title slide should have large, bold text (36pt+)
- Content should be readable (minimum 18pt font)
- Maintain consistent spacing and alignment
- Don't overcrowd slides

Final Step
```

```
Store the completed artifact in a variable called `artifact`.
EOF
```

Document Skill Guide

```
Shell
cat > src/skills/document/skill.md << 'EOF'
# Document Creation Skill

## Overview
You are creating a document using the `DocumentArtifact` class.

## Basic Usage
```python
# Create a new document
artifact = DocumentArtifact("my_document")

# Add paragraphs
artifact.add_paragraph("Title", style="Heading 1")
artifact.add_paragraph("This is the introduction.",
style="Normal")
artifact.add_paragraph("Section 1", style="Heading 2")
artifact.add_paragraph("Section content goes here.",
style="Normal")
```

Quality Standards
- Use proper heading hierarchy (Heading 1, Heading 2, etc.)
- Maintain consistent formatting
- Keep paragraphs focused and readable

Final Step
Store the completed artifact in a variable called `artifact`.
EOF
```

Part 10: Testing Your System

Chapter 25: Why Test?

Before packaging everything in Docker, let's verify the formula engine works correctly. Tests catch bugs early and give you confidence the system behaves as expected.

Think of tests as a safety net. When you make changes later, running the tests tells you immediately if you broke something.

Chapter 26: Creating the Test File

Create the test file:

```shell
cat > tests/test_formula_engine.py << 'EOF'
"""Tests for the formula evaluation engine."""

import pytest
from src.artifact_tool.spreadsheet import SpreadsheetArtifact


def test_basic_sum():
    """Test that SUM formula calculates correctly."""
    artifact = SpreadsheetArtifact("test")
    sheet = artifact.get_sheet("Sheet1")

    sheet.set_cell("A1", value=10)
    sheet.set_cell("A2", value=20)
    sheet.set_cell("A3", formula="=SUM(A1:A2)")

    artifact.recalculate()

    assert sheet["A3"].number_value == 30.0


def test_basic_average():
    """Test that AVERAGE formula calculates correctly."""
```

```
    artifact = SpreadsheetArtifact("test")
    sheet = artifact.get_sheet("Sheet1")

    sheet.set_cell("B1", value=100)
    sheet.set_cell("B2", value=200)
    sheet.set_cell("B3", formula="=AVERAGE(B1:B2)")

    artifact.recalculate()

    assert sheet["B3"].number_value == 150.0


def test_string_values():
    """Test that string values are stored correctly."""
    artifact = SpreadsheetArtifact("test")
    sheet = artifact.get_sheet("Sheet1")

    sheet.set_cell("A1", value="Hello")

    assert sheet["A1"].string_value == "Hello"


def test_bool_not_treated_as_int():
    """Test that booleans aren't confused with integers.

    In Python, bool is a subclass of int (True == 1, False == 0).
    We need to check for bool BEFORE int to handle this
correctly.
    """
    artifact = SpreadsheetArtifact("test")
    sheet = artifact.get_sheet("Sheet1")

    sheet.set_cell("A1", value=True)

    # Should be stored as bool, not as number 1
    assert sheet["A1"].bool_value == True
```

```
    assert not sheet["A1"].HasField("number_value")
EOF
```

Chapter 27: Running Tests Locally

To run the tests locally (before Docker):

Shell

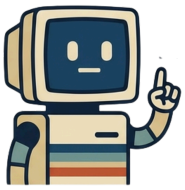
```
# First, set up a virtual environment
python3 -m venv venv
source venv/bin/activate

# Install dependencies
pip install -r requirements.txt

# Set PYTHONPATH so imports work correctly
export PYTHONPATH=$(pwd):$(pwd)/src/generated

# Run tests
python -m pytest tests/ -v
```

You should see output like:

None

```
tests/test_formula_engine.py::test_basic_sum PASSED
tests/test_formula_engine.py::test_basic_average PASSED
tests/test_formula_engine.py::test_string_values PASSED
tests/test_formula_engine.py::test_bool_not_treated_as_int PASSED
========================= 4 passed =========================
```

Congratulations! Your formula engine works correctly.



TIP: If tests fail, the error messages tell you exactly what went wrong. Fix the issue and run again. This is much faster than debugging inside Docker.

Part 11: Docker: Packaging Everything

Chapter 28: Understanding the Dockerfile

A Dockerfile is a recipe for building a Docker image. Each line is an instruction.

Shell

```
cat > Dockerfile << 'EOF'
FROM ubuntu:22.04

ENV DEBIAN_FRONTEND=noninteractive
ENV PYTHONUNBUFFERED=1

# Install system dependencies
RUN apt-get update && apt-get install -y \
    python3 \
    python3-venv \
    python3-pip \
    libreoffice \
    poppler-utils \
    supervisor \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Install Node.js 22.x (Ubuntu's default nodejs is too old)
RUN curl -fsSL https://deb.nodesource.com/setup_22.x | bash - \
    && apt-get install -y nodejs

# Set working directory
WORKDIR /app
```

```
# Set PYTHONPATH so imports work correctly
# - /app: for application imports like "from
src.artifact_tool..."
# - /app/src/generated: for generated protobuf internal imports
like "from proto..."
ENV PYTHONPATH=/app:/app/src/generated

# Copy and install Python dependencies
COPY requirements.txt .
RUN pip3 install --no-cache-dir -r requirements.txt

# Copy and install JavaScript dependencies
COPY js_pipeline/package*.json ./js_pipeline/
RUN cd js_pipeline && npm install

# Copy source code
COPY src/ ./src/
COPY js_pipeline/ ./js_pipeline/

# Copy test files
COPY tests/ ./tests/

# Copy supervisor configuration
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

# Create output directory
RUN mkdir -p /app/output

# Expose port for JS pipeline
EXPOSE 3000

# Run supervisor (manages both services)
CMD ["/usr/bin/supervisord", "-c",
"/etc/supervisor/conf.d/supervisord.conf"]
EOF
```

Let's understand each section:

`FROM ubuntu:22.04`: Start with Ubuntu 22.04 as the base.

`ENV ...`: Set environment variables. `DEBIAN_FRONTEND=noninteractive` prevents installation prompts.

`RUN apt-get ...`: Install system packages.

`WORKDIR /app`: Set the working directory for subsequent commands.

`COPY requirements.txt .`: Copy just the requirements file first.

`RUN pip3 install ...`: Install Python packages. By copying requirements.txt separately, this layer gets cached—changes to your code won't trigger a reinstall. This is a Docker best practice for faster builds.

`EXPOSE 3000`: Document that port 3000 is used (doesn't actually open it).

`CMD [...]`: The default command when the container starts.

Chapter 29: Supervisor Configuration

Supervisor manages multiple processes inside the container:

```shell
cat > supervisord.conf << 'EOF'
[supervisord]
nodaemon=true
logfile=/var/log/supervisord.log
pidfile=/var/run/supervisord.pid

[program:node_pipeline]
command=node /app/js_pipeline/server.js
directory=/app/js_pipeline
autostart=true
autorestart=true
stderr_logfile=/var/log/node_pipeline.err.log
stdout_logfile=/var/log/node_pipeline.out.log
environment=NODE_ENV=production
```

```
[program:python_agent]
command=python3 /app/src/agent/main.py
directory=/app
autostart=false
autorestart=false
stderr_logfile=/var/log/agent.err.log
stdout_logfile=/var/log/agent.out.log
environment=PYTHONPATH=/app:/app/src/generated
EOF
```

nodaemon=true: Run in foreground (Docker expects this).

autostart=true: The Node server starts automatically.

autostart=false: The Python agent waits for explicit start (so we can pass arguments).

> **Note:** This configuration is designed for the Docker environment. The Node.js presentation server starts automatically, but the Python agent does not—this allows you to pass custom tasks via command line arguments. When you run the container, you'll override the default CMD to run the agent directly (see Chapter 33).

Chapter 30: Docker Compose

Docker Compose simplifies running containers. Note that this is primarily useful for development and testing—for running actual tasks, you'll use the direct docker run command shown in Chapter 33.

Shell

```
cat > docker-compose.yml << 'EOF'
version: '3.8'

services:
  orion-agent:
    build: .
    container_name: orion-agent
```

```
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
    volumes:
      - ./output:/app/output
    ports:
      - "3000:3000"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 10s
      timeout: 5s
      retries: 3
EOF
```

volumes: Maps your local output folder to `/app/output` in the container. Files saved there appear on your host machine.

ports: Maps port 3000 on your computer to port 3000 in the container.

healthcheck: Verifies the presentation server is running correctly.

Chapter 31: Environment Variables File

Create a template for your environment variables:

Shell

```
cat > .env << 'EOF'
# OpenAI API Key (required)
# Get yours at: https://platform.openai.com/api-keys
OPENAI_API_KEY=your_api_key_here
EOF
```

IMPORTANT: Never commit your `.env` file to Git—it contains secrets. The `.gitignore` we created earlier already excludes it.

You can either:

1. Edit this file and add your real API key, OR

2. Export the key directly in your terminal (shown in the next chapter)

Part 12: Running ORION

Chapter 32: Building the Image

First, install the JavaScript dependencies locally (this also generates the package-lock.json file that Docker will use):

```shell
cd js_pipeline
npm install
cd ..
```

Now build the Docker image:

```shell
docker build -t orion-agent:1.0 .
```

This takes several minutes the first time—Docker is downloading base images, installing packages, and copying files. Subsequent builds are faster due to caching.

You should see output ending with something like:

```none
Successfully built abc123def456
Successfully tagged orion-agent:1.0
```

Chapter 33: Running the Agent

Set your OpenAI API key:

Shell

```
export OPENAI_API_KEY="sk-your-key-here"
```

IMPORTANT: Replace `sk-your-key-here` with your actual OpenAI API key. You can get one at platform.openai.com.

Now run ORION with a task:

Shell

```
docker run --rm \
  -e OPENAI_API_KEY="$OPENAI_API_KEY" \
  -v "$(pwd)/output:/app/output" \
  -p 3000:3000 \
  orion-agent:1.0 \
  bash -c '
    # Start Node server in background
    node /app/js_pipeline/server.js &

    # Wait for server to be ready (up to 30 seconds)
    echo "Waiting for presentation server..."
    for i in $(seq 1 30); do
      if curl -s http://localhost:3000/health > /dev/null 2>&1;
then
        echo "Server ready!"
        break
      fi
      sleep 1
    done

    # Run the agent
    python3 /app/src/agent/main.py "Create a spreadsheet showing
quarterly sales for Q1-Q4"
  '
```

Let's break down what's happening:

- `docker run`: Start a container.
- `--rm`: Remove the container when it exits.
- `-e OPENAI_API_KEY="$OPENAI_API_KEY"`: Pass your API key to the container.
- `-v "$(pwd)/output:/app/output"`: Mount your output folder so you can see the results.
- `-p 3000:3000`: Map port 3000 for the presentation server.
- `orion-agent:1.0`: The image to run.
- `bash -c "..."`: Run a shell command that starts the Node server in the background, waits for it to be ready via health check, then runs the Python agent.

Chapter 34: Watching It Work

When you run ORION, you'll see output like:

```
None
Starting ORION Agent with task: Create a spreadsheet showing
quarterly sales

--- Iteration 1 ---
Generating code...
Generated code:
artifact = SpreadsheetArtifact("quarterly_sales")
sheet = artifact.get_sheet("Sales")
sheet.set_cell("A1", value="Quarter")
...

Executing code...
Running QA verification...
QA PASSED!

Success! Output saved to: /app/output/quarterly_sales.xlsx
```

Check your output folder—you should see the generated file!

If QA fails, you'll see the feedback and watch ORION try again:

```
None
QA FAILED. Feedback: The column headers are present but some
cells appear empty...

--- Iteration 2 ---
Generating code...
```

Chapter 35: Running Tests in Docker

You can also run the test suite inside the Docker container to verify everything works in the same environment where the agent runs:

```
Shell
docker run --rm orion-agent:1.0 python3 -m pytest /app/tests/ -v
```

You should see:

```
None
tests/test_formula_engine.py::test_basic_sum PASSED
tests/test_formula_engine.py::test_basic_average PASSED
tests/test_formula_engine.py::test_string_values PASSED
tests/test_formula_engine.py::test_bool_not_treated_as_int PASSED
========================= 4 passed =========================
```

This confirms that the formula engine works correctly inside Docker, with all the dependencies properly installed.

> **Note:** This command overrides the default container startup, so the Node.js server won't be running. This is fine for the current tests which don't need it. If you add tests that create presentations, you'll need to start the Node server first (similar to Chapter 33).

Part 13: Extending ORION

Chapter 36: Adding a New Artifact Type

Want to add a new document type? Here's the process:

1. Create a `.proto` file defining the data model
2. Compile it to generate Python code
3. Create an artifact class that inherits from `BaseArtifact`
4. Implement the export method
5. Add it to `local_vars` in the agent
6. Create a skill guide
7. Update skill selection logic

Example: Adding a Calendar Artifact

First, the proto file:

protobuf

```protobuf
// src/proto/calendar.proto
syntax = "proto3";

package orion.artifact;

message Calendar {
    string artifact_id = 1;
    repeated Event events = 2;
}

message Event {
    string title = 1;
    string start_time = 2;
    string end_time = 3;
    optional string description = 4;
}
```

Then the artifact class, export method, skill guide... you know the pattern now!

Chapter 37: Improving Security

The current implementation uses `exec()` which runs arbitrary code. This is fine for trusted environments where Docker provides isolation. For production systems:

1. Use RestrictedPython to limit what code can do
2. Run each task in a fresh, throwaway container
3. Add time limits and resource limits
4. Sandbox network access

Part 14: Troubleshooting

Chapter 38: Common Issues and Solutions

Here are solutions to problems you might encounter:

"LibreOffice conversion timed out"

Cause: LibreOffice is taking too long to convert a document to PDF.

Solutions:

- Restart the Docker container (LibreOffice sometimes gets stuck)
- Increase the timeout in `render_and_verify()`
- Simplify the document (fewer elements = faster conversion)

```shell
# Restart the container
docker stop orion-agent
docker run ... # (your run command)
```

"Cannot connect to presentation server"

Cause: The Node.js server isn't running when trying to create a presentation.

Solutions:

- Make sure you're using the full docker run command from Chapter 33
- Check that port 3000 isn't already in use on your machine
- Verify the Node server started: look for "Presentation pipeline listening on port 3000" in the output

```shell
# Check if port 3000 is in use
lsof -i :3000
```

"ModuleNotFoundError: No module named 'src'"

Cause: PYTHONPATH isn't set correctly.

Solutions:

- When running locally, set PYTHONPATH: `export PYTHONPATH=$(pwd)`
- In Docker, this is handled automatically by the Dockerfile
- Make sure you're running from the `orion-agent` directory

"protoc: command not found"

Cause: Protocol Buffers compiler isn't installed.

Solution:

```shell
brew install protobuf
```

"OPENAI_API_KEY environment variable not set"

Cause: The API key wasn't passed to the container.

Solutions:

- Set the key before running: `export OPENAI_API_KEY="sk-..."`
- Make sure you're using `-e OPENAI_API_KEY="$OPENAI_API_KEY"` in your docker run command
- Check for typos in the variable name

Tests pass locally but fail in Docker

Cause: Environment differences between local and Docker.

Solutions:

- Run tests in Docker to ensure consistency: `docker run --rm orion-agent:1.0 python3 -m pytest /app/tests/ -v`
- Check that all dependencies are in requirements.txt
- Verify the PYTHONPATH is set correctly in the Dockerfile

"No images were generated from the document"

Cause: The PDF conversion succeeded but image conversion failed.

Solutions:

- Verify poppler-utils is installed (it provides pdftoppm)
- Check the document isn't empty
- Look for error messages in the QA feedback

Docker build is slow

Cause: Docker isn't caching layers effectively.

Solutions:

- Make sure `.dockerignore` exists and includes unnecessary files
- Don't modify `requirements.txt` or `package.json` unless necessary
- Use `docker build --cache-from` for CI/CD pipelines

Running locally without Docker

If you want to run ORION locally without Docker (for faster development iteration):

```shell
# Terminal 1: Start the Node server
cd js_pipeline
npm install  # First time only
node server.js

# Terminal 2: Run the agent
cd ..  # Back to orion-agent directory
source venv/bin/activate
export PYTHONPATH=$(pwd)
export OPENAI_API_KEY="sk-..."
python3 src/agent/main.py "Your task here"
```

Note: You'll need LibreOffice and poppler installed locally (`brew install --cask libreoffice` and `brew install poppler`).

Glossary

Agent: A program that autonomously takes actions to achieve a goal.

API (Application Programming Interface): A way for programs to communicate with each other.

Artifact: In ORION, a document being created (spreadsheet, presentation, document).

Base64: A way to encode binary data as text.

Container: A lightweight, isolated environment for running software.

Docker: A platform for running containers.

Environment Variable: A variable set in the operating system, accessible to programs.

Homebrew: A package manager for macOS.

JSON (JavaScript Object Notation): A text format for structured data.

Package Manager: A tool that installs and manages software packages.

Protocol Buffers (Protobufs): A language-neutral format for defining data structures.

QA (Quality Assurance): The process of checking that something meets requirements.

Terminal: A text-based interface for controlling your computer.

Vision Model: An AI model that can analyze images.

Conclusion

You now understand every piece of Project ORION:

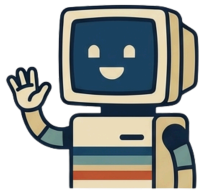
- What each technology is and why it was chosen
- How data flows through the system
- What each file does and why it exists
- How to run, modify, and extend the system

This isn't just about ORION—these patterns appear in AI systems everywhere. Protocol Buffers, Docker containers, agent loops, vision verification... these are the building blocks of modern AI infrastructure.

You've just built something real. Something that works. Something you understand from top to bottom.

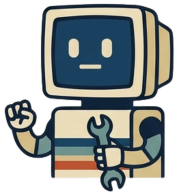
The next step is yours: go build something amazing with it.

> **Remember:** The best way to learn is to experiment. Try changing things. Break stuff. Fix it. That's how you really understand how systems work.



Welcome to the future of computing. Have fun!

Errata & Corrections (Master Edition)



This Master Edition includes the following corrections from the original document:

Fix #1: JavaScript Server Layout Error (Page 59)

Original (broken):

javascript

```javascript
if (artifact.slide_width && artifact.slide_height) {
    pres.layout = "LAYOUT_CUSTOM";
    pres.defineLayout({...});
}
```

Corrected:

javascript

```javascript
if (artifact.slide_width && artifact.slide_height) {
    pres.defineLayout({...});
    pres.layout = "CUSTOM";
}
```

The layout must be defined before it can be assigned. Also changed `"LAYOUT_CUSTOM"` to `"CUSTOM"` to match the defined layout name.

Fix #2: Presentation Artifact SameFileError (Page 38)

Original (broken):

python

```python
shutil.copy(server_path, path)
```

Corrected:

python

```python
abs_server_path = os.path.abspath(server_path)
abs_dest_path = os.path.abspath(path)
if abs_server_path != abs_dest_path:
    shutil.copy(server_path, path)
```

Added check to prevent copying a file onto itself when source and destination paths resolve to the same location.

Master Edition prepared December 14, 2025