R for the Built Environment

Paul Govan

2026-04-22

Table of contents

Preface 8
Why R? 8
What You'll Learn 9
How This Book is Structured 9
Prerequisites 10
Installation 10
How to Use This Book 10
Where This is Headed 10

I Front Matter 12

Motivation 13
The Problem 13
The Opportunity 13
The Solution 13
Who This Book Is For 14

Acknowledgements 15
How to Cite 15

License 17
Book Content 17
AutoDeskR Package 17
AutoDesk Platform Services 18

II Getting Started 19

Getting Started 20
System Requirements 20
Step 1: Create an AutoDesk Account 20
Step 2: Register an APS Application 20
Step 3: Store Credentials Securely 21
Step 4: Install AutoDeskR 21
Step 5: Your First Workflow 21
What's Next 23

III AutoDesk Platform Services 24

1 Authentication 25
1.1 Store Credentials Safely . . . 25
1.2 Get an Access Token . . . 26
1.3 Scopes — Only Ask for What You Need . . . 26
1.4 Rate Limits . . . 27

2 Data Management 29
2.1 Create a Bucket . . . 29
2.2 Upload a File . . . 30
2.2.1 Big Files (> 100 MB) . . . 31
2.3 List Buckets and Objects . . . 31
2.4 Delete Objects and Buckets . . . 32

3 Model Derivative 34
3.1 Translate to OBJ . . . 34
3.2 Translate to STL . . . 36
3.3 Extract Metadata from a File . . . 36

4 Design Automation 39
4.1 Convert a DWG to PDF . . . 39
4.2 Poll for Completion . . . 40
4.3 Batch Conversion . . . 41

5 Reality Capture 42
5.1 Before You Start . . . 42
5.2 Step 1: Authenticate . . . 43
5.3 Step 2: Create a Photoscene . . . 43
5.4 Step 3: Upload Images . . . 43
5.5 Step 4: Start Processing . . . 44
5.6 Step 5: Wait for It . . . 44
5.6.1 Option A — Block until done (recommended) . . . 44
5.6.2 Option B — Manual polling . . . 45
5.7 Step 6: Download the Output . . . 45

6 Viewer 46
6.1 Launch the Viewer . . . 46
6.2 Embed in a Shiny App . . . 47
6.2.1 Viewer in a Tab Panel . . . 47
6.2.2 Viewer + Sensor Dashboard . . . 48
6.3 What's Displayed . . . 48

IV 3D Geometry & Meshes 49

7 Reading OBJ and STL Meshes 50
7.1 Prerequisites . . . 50
7.2 Getting the OBJ File . . . 50

7.3 Reading with Rvcg (recommended for analysis) . . . 51
7.4 Reading with rgl (for interactive viewing) . . . 51
7.5 Reading STL . . . 52
7.6 Quick Inspection . . . 52

8 Mesh Metrics **53**
8.1 Surface Area . . . 53
8.2 Volume . . . 53
8.3 Bounding Box . . . 53
8.4 Centroid . . . 54
8.5 Convex Hull Volume . . . 54
8.6 Summary Table . . . 54
8.7 Visualising the Metrics . . . 55

9 3D Visualisation **57**
9.1 Interactive 3D with rgl . . . 57
9.1.1 Embedding in HTML Output . . . 57
9.2 Colouring by Height . . . 58
9.3 Rendered Images with rayshader . . . 58
9.4 Depth of Field . . . 59
9.5 Quick Comparison: rgl vs. rayshader . . . 59

10 Point Cloud Analysis **60**
10.1 Getting the Point Cloud . . . 60
10.2 Reading the Point Cloud . . . 60
10.3 Height Statistics . . . 61
10.4 Height Distribution . . . 61
10.5 Canopy Height Model . . . 62
10.6 Point Density Grid . . . 62
10.7 Voxel Volume . . . 62
10.8 Summary Table . . . 63

11 Mesh Comparison **64**
11.1 Loading Two Meshes . . . 64
11.2 Smoothing the As-Built Mesh . . . 64
11.3 Surface Distance . . . 64
11.4 Deviation Colour Map . . . 65
11.5 Summary Statistics . . . 65
11.6 Deviation Histogram . . . 66
11.7 Cumulative Distribution . . . 67

V DWG & DXF Analytics **68**

12 Layer Structure Analysis **69**
12.1 Prerequisites . . . 69
12.2 Authenticate and Translate . . . 69
12.3 The Object Tree . . . 70
12.4 Flatten the Tree . . . 70

12.5 Extract Properties with getData() . . . 70
12.6 Summarise by Layer . . . 71
12.7 Area by Layer — Bar Chart . . . 72
12.8 Objects per Layer — Dot Plot . . . 72

13 Attribute Extraction **74**
13.1 Custom Properties . . . 74
13.2 Block Attributes . . . 74
13.3 Pivot to Wide Format . . . 75
13.4 Join Attributes to Layer Data . . . 75
13.5 Floor Area by Zone and Construction Type . . . 76
13.6 Visualising Custom Attributes . . . 76
13.7 Export . . . 77

14 Cross-Drawing Comparison **78**
14.1 Upload Multiple Drawings . . . 78
14.2 Extract Layer Sets . . . 78
14.3 What Changed? . . . 79
14.4 Presence Matrix . . . 79
14.5 Layer Presence Heatmap . . . 79
14.6 Object Count Changes . . . 80
14.7 Change Waterfall Chart . . . 81

15 Automated Drawing Reports **83**
15.1 A gt Summary Table . . . 83
15.2 Parameterised Reports . . . 84
15.3 Render Programmatically . . . 84
15.4 Scheduling Reports . . . 85

VI Digital Twins **87**

16 AutoDesk Tandem Overview **88**
16.1 Twin vs. BIM Model . . . 88
16.2 Authentication . . . 89
16.3 Listing Facilities . . . 89
16.4 Facility Details . . . 89
16.5 Looking Up an Element . . . 90

17 Linking Sensor Streams **91**
17.1 Listing Streams for a Facility . . . 91
17.2 Fetching Time-Series Readings . . . 92
17.3 Plotting Readings . . . 92
17.4 Joining Sensor Data to Model Metadata . . . 93
17.5 Fetching All Streams at Once . . . 94
17.6 Multi-Stream Overview Plot . . . 94

18 Live Dashboards **96**
18.1 Required Packages . . . 96

18.2 Helper: Fetch Stream Readings . . . 96
18.3 App Startup . . . 97
18.4 UI Layout . . . 98
18.5 Server Logic . . . 98
18.6 Deploying to shinyapps.io . . . 99

VII AI Integration **100**

19 MCP Server **101**
19.1 Proposed Tools . . . 101
19.2 Setting Up a Local MCP Server . . . 102
19.3 Connecting to Claude Desktop . . . 103
19.4 Example Agent Conversations . . . 103
19.5 Security Considerations . . . 104

VIIIReference **105**

20 Case Study: DWG to Shiny Dashboard **106**
20.1 Step 1: Authenticate with Combined Scopes . . . 106
20.2 Step 2: Create a Bucket and Upload the DWG . . . 106
20.3 Step 3: Translate to SVF . . . 107
20.4 Step 4: Extract Layer Metadata . . . 107
20.5 Step 5: Visualise with ggplot2 . . . 108
20.6 Step 6: Build the Shiny Dashboard . . . 109
20.7 Summary . . . 110

21 Function Reference **111**
21.1 Authentication . . . 111
21.2 Data Management . . . 111
21.3 Model Derivative . . . 112
21.4 Design Automation . . . 112
21.5 Reality Capture . . . 113
21.6 Viewer . . . 113
21.7 Scope Quick Reference . . . 114
21.8 Common Patterns . . . 114
21.8.1 Base-64 encode a URN . . . 114
21.8.2 Poll until a job finishes . . . 114

22 Supporting Packages **116**
22.1 httr2 . . . 116
22.2 jsonlite . . . 116
22.3 Rvcg . . . 117
22.4 rgl . . . 117
22.5 geometry . . . 117
22.6 rayshader . . . 118
22.7 lidR . . . 118
22.8 dplyr . . . 118

22.9 tidyr . . . 119
22.10 ggplot2 . . . 119
22.11 gt . . . 119
22.12 shiny . . . 120
22.13 dygraphs . . . 120
22.14 xts . . . 121
22.15 quarto . . . 121

23 Troubleshooting **122**
23.1 Error Code Quick Reference . . . 122
23.2 Diagnosing Any Error . . . 123
23.3 Installation & Setup . . . 123
23.4 Authentication . . . 124
23.5 Data Management . . . 125
23.6 Model Derivative . . . 126
23.7 Design Automation . . . 127
23.8 Reality Capture . . . 127
23.9 Viewer . . . 128
23.10 3D Geometry (rgl / Rvcg) . . . 128
23.11 Digital Twins (Tandem) . . . 129
23.12 Getting More Help . . . 129

References **130**

Preface

The built environment generates an enormous amount of data. Architectural drawings, structural models, survey point clouds, energy performance figures, sensor readings from every floor of every building. Most of it sits inside proprietary formats that R has historically had no good way to reach. *R for the Built Environment* changes that.

This book uses the AutoDeskR R package (Govan 2024) and the AutoDesk Platform Services (APS) cloud API as the primary bridge to that data, translating design files, extracting geometry, connecting live sensor streams, and embedding interactive 3D models in Shiny dashboards. AutoDesk is the dominant platform in the AEC industry, so it's a practical place to start. But the analytical techniques here, mesh analysis, layer analytics, digital twin patterns, MCP tool exposure, apply to any BIM or CAD data, wherever it comes from.

Whether you're a data scientist who just inherited a folder of DWG files, or a BIM manager who wants to automate something that currently takes three software packages and a prayer, this book is for you.

Why R?

Fair question. Most AEC professionals use Python, .NET, or whatever scripting language AutoCAD yells at you in. So why drag R into this?

Because the audience isn't architects, it's analysts. The real use case here is the data scientist who gets handed a folder of DWG files and told to "make a dashboard out of this." That person lives in R. They know dplyr, they know ggplot2, they know Shiny. AutoDeskR gives them a bridge to the CAD world without forcing them to learn a whole new stack.

Because the Python SDK is for building applications. It's great if you're writing production software. But if you want to pull geometry data, run some summary statistics, and produce a polished report with a table and a plot, R's ecosystem (ggplot2, gt, Quarto) is still unmatched. The wrapper isn't competing with the Python SDK; it's serving a different job.

Because thin wrappers encode real expertise. Yes, a sophisticated user could call the APS REST API directly with `httr2`. But do they know that the object URN needs to be Base64-encoded before passing it to the Model Derivative API? That tokens expire after exactly 3600 seconds? That bucket names are globally unique across *all* APS applications? AutoDeskR quietly handles all of that so you can focus on the analysis, not the plumbing.

Because API drift is a fact of life, not a dealbreaker. AutoDesk updates its APIs. So do Google, AWS, and everyone else. Staying close to the API surface means fewer moving parts to break, and direct `httr2` calls would have exactly the same maintenance problem with more boilerplate.

Because the free tier is for experimenting, and that's the whole point. Once you've prototyped something worth running in production, you're almost certainly inside an organisation that's already paying for AutoDesk products, and API access comes with the subscription.

What You'll Learn

Each chapter covers a different piece of the BIM and CAD analytics ecosystem:

- **Authentication** — grab an OAuth token in one line and never think about it again
- **Data Management** — upload files to the cloud, wrangle buckets, and pull objects back down
- **Model Derivative** — translate design files into OBJ, STL, and SVF; extract geometry and metadata
- **Design Automation** — run DWG-to-PDF conversion in the cloud, no AutoCAD required
- **Reality Capture** — turn a set of overlapping photos into a 3D mesh
- **Viewer** — embed a live 3D model viewer in a Shiny app with two lines of code
- **3D Geometry & Meshes** — import, measure, and visualise translated mesh files
- **DWG & DXF Analytics** — parse layer structure, extract attributes, compare drawing revisions
- **Digital Twins** — link live sensor data to BIM elements via AutoDesk Tandem
- **MCP Server** — expose AutoDeskR as an AI agent tool so LLMs can query BIM models directly

How This Book is Structured

The book is organised into eight parts. You don't have to read them in order, each chapter is reasonably self-contained, but the parts do build on each other if you want the full picture.

Part	Chapters	What you'll do
Front Matter	Motivation, Acknowledgements	Understand the why
Getting Started	Getting Started	Install the package and make your first API call
AutoDesk Platform Services	Authentication → Viewer	The six main APS APIs, start to finish
3D Geometry & Meshes	Reading → Visualisation → Point Clouds → Comparison	Import, measure, and compare translated mesh files
DWG & DXF Analytics	Layer Analysis → Attributes → Comparison → Reports	Extract meaning from drawing data
Digital Twins	Tandem Overview → Sensor Streams → Live Dashboards	Join live sensor readings to BIM elements
AI Integration	MCP Server	Expose AutoDeskR as a tool for AI agents
Reference	Case Study, Function Reference, Troubleshooting	Worked example, lookup tables, and help

Prerequisites

To follow along you'll need:

- **R** (version 4.0 or later) and, optionally, RStudio
- A free **AutoDesk account** with an APS application registered at aps.autodesk.com.The Getting Started chapter walks you through it step by step
- An internet connection (all the action happens in AutoDesk's cloud)

No prior CAD or BIM experience assumed. If you're comfortable running an R script, you're ready.

Installation

Install the stable release from CRAN:

```
install.packages("AutoDeskR")
```

Or grab the development version from GitHub if you like living on the edge:

```
devtools::install_github("paulgovan/AutoDeskR")
```

How to Use This Book

New to APS? Start with Getting Started and work through the chapters in order, each one builds on the previous. Already know the APS basics? Jump straight to whatever chapter covers your use case. The Troubleshooting chapter is always there when something goes sideways.

All code examples are copy-paste ready and work with `#| eval: false` so they won't execute during book rendering but will run just fine in your R session.

Where This is Headed

This book covers a lot of ground, but the APS API catalogue is bigger than any one book. Here's what's on the horizon.

Near-term package additions — AutoDeskR currently wraps the six core APS APIs. The next wave of functions will cover the richer ACC project hierarchy (Hubs → Projects → Folders → Items → Versions), which is required for working with BIM 360 and ACC projects where files live in managed folder trees rather than flat OSS buckets. Alongside that: Webhooks (so your pipelines can react to events instead of polling), Construction Issues (live project health data in R), and Account Admin (bulk user and project management via script instead of clicking through a web interface a hundred times).

Advanced Design Automation — `makePdf()` is just the beginning. APS Design Automation also supports Revit, 3ds Max, Inventor, and Fusion, meaning you can run custom scripts against any of those engines from R without installing the software locally. Generalised `submitWorkItem()` and `listEngines()` functions are planned to unlock that.

Longer-term — Model Coordination (clash detection and model sets, once the ACC hierarchy is in place), Forma (AutoDesk's AEC industry cloud), and Cost Management (budget and financial data for project dashboards) are all on the radar. And as IFC, Speckle, and other open BIM platforms mature, they belong here too. The guiding rule: if it takes built environment data as its primary input, it belongs in this book.

Track progress and file requests at github.com/paulgovan/AutoDeskR.

Part I

Front Matter

Motivation

The Problem

The built environment is full of data. Layer structure, geometry, material properties, object attributes, energy performance figures, sensor readings linked to physical elements, all of it sits inside proprietary file formats that the rest of the data science world can't easily reach. The tools that can read these files (AutoCAD, Revit, Rhino) are desktop applications built for design, not analysis. They're not going to produce a ggplot2 chart or feed a Shiny dashboard.

The result is an unnecessary gap. Engineers and architects generate rich, spatially-grounded data every day. Data scientists who could turn that data into insight have no good way to access it. Both teams end up frustrated, exporting things to CSV by hand and losing half the structure in the process.

The Opportunity

R has quietly become the lingua franca for quantitative work in architecture, engineering, and construction (AEC). Structural engineers use it to crunch sensor data. Sustainability consultants model energy performance in it. BIM managers audit model quality across large portfolios with it. And Shiny brings all of that to non-technical stakeholders through interactive dashboards.

What's been missing is a clean bridge between R's data science ecosystem and the 3D model data locked inside CAD files. A DWG file isn't just a drawing, it's geometry, layer structure, material properties, and embedded attributes that are genuinely interesting to analyse. An as-built point cloud from a drone survey can be compared against a design mesh to quantify construction deviations. A BIM model wired to live sensor data becomes a digital twin that a Shiny app can query in real time.

The Solution

This book builds that bridge using the AutoDesk Platform Services (APS) cloud API as its primary engine. AutoDesk is the dominant platform in the AEC industry, and APS can translate design files between dozens of formats, extract rich geometry and metadata from BIM models, automate DWG processing at scale, and render 3D models interactively in a browser — all without a local AutoCAD or Revit installation.

The catch is that APS speaks REST. Every operation means authenticated HTTP requests, base64-encoded URNs, polling asynchronous job queues, and parsing deeply nested JSON. AutoDeskR (Govan 2024) wraps all of that in idiomatic R functions. Authentication, encoding, polling —

handled. A workflow that would otherwise mean dozens of raw HTTP calls comes down to a handful of lines:

```r
library(AutoDeskR)

token  <- getToken(id, secret, scope = "data:read data:write bucket:create")
bucket <- makeBucket(token, "my-project-bucket")
upload <- uploadFile(token, "model.dwg", bucket$bucketKey)
job    <- translateFile(token, upload$objectId, "svf")
```

The goal isn't to paper over APS entirely. Understanding what's happening under the hood makes you more effective. AutoDeskR just removes the boilerplate so you can spend your time on the analysis, not the plumbing.

Who This Book Is For

This book is for R users who work with or alongside CAD and BIM data and want to automate, analyse, or visualise it without leaving the R environment. The techniques here, mesh analysis, layer analytics, digital twin patterns, point cloud processing, apply to BIM and CAD data broadly, not just data that came from AutoDesk software. AutoDesk is where we start because that's where most of the industry's data currently lives, but the analytical mindset carries over wherever the files come from.

No prior experience with APS, AutoDesk, or the AEC industry is assumed. If you can install an R package and run a script, you have everything you need.

Acknowledgements

R for the Built Environment would not exist without the work of many individuals and organisations who generously share their time, code, and knowledge.

AutoDesk designed and maintains the AutoDesk Platform Services (APS) APIs and provides extensive developer documentation, sample code, and a free tier that makes experimentation accessible to individuals and small teams. The quality of their API design made it possible to wrap APS in a concise R interface.

The R community — CRAN maintainers, package authors, and the broader community of practitioners who publish tutorials, answer questions, and review code provides the foundation on which AutoDeskR is built. In particular, the httr2 and jsonlite packages handle the HTTP and JSON plumbing that underlies every API call in this package.

Posit (formerly RStudio) develops and maintains Quarto, the publishing system used to produce this book, as well as the devtools, usethis, and roxygen2 packages that are central to R package development.

Contributors to AutoDeskR who have filed issues, suggested improvements, and submitted pull requests on GitHub have shaped the package into what it is today. Their feedback has been invaluable.

Finally, thank you to everyone who has read earlier drafts of this material and offered corrections and suggestions.

How to Cite

If this book or the AutoDeskR package is useful in your work, please consider citing it.

Citing the book:

```
@book{govan2026builtenv,
  title     = {R for the Built Environment},
  author    = {Paul Govan},
  year      = {2026},
  url       = {https://paulgovan.github.io/AutoDeskR-Book/}
}
```

APA: Govan, P. (2026). *R for the Built Environment*. Retrieved from https://paulgovan.github.io/AutoDeskR-Book/

Citing the package:

```
@manual{R-AutoDeskR,
  title  = {AutoDeskR: An Interface to the AutoDesk Platform Services API},
  author = {Paul Govan},
  year   = {2024},
  note   = {R package},
  url    = {https://github.com/paulgovan/AutoDeskR}
}
```

APA: Govan, P. (2024). *AutoDeskR: An Interface to the AutoDesk Platform Services API* [R package]. https://github.com/paulgovan/AutoDeskR

You can also get the package citation directly from R:

```
citation("AutoDeskR")
```

License

Book Content

The text, code examples, and figures in *R and the AutoDesk Platform* are licensed under the **Creative Commons Attribution 4.0 International License (CC BY 4.0)**.

You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially

Under the following terms:

- **Attribution** — You must give appropriate credit to the original author (Paul Govan), provide a link to the license, and indicate if changes were made.

Full license text: https://creativecommons.org/licenses/by/4.0/

AutoDeskR Package

The AutoDeskR R package is distributed under the **MIT License**.

```
MIT License

Copyright (c) Paul Govan

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
```

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

AutoDesk Platform Services

Use of the AutoDesk Platform Services (APS) APIs is subject to AutoDesk's own Terms of Service and Developer Terms. This book is an independent work and is not affiliated with or endorsed by AutoDesk, Inc.

Part II

Getting Started

Getting Started

Five steps stand between you and your first successful API call. By the end of this chapter you'll have credentials stored safely, AutoDeskR installed, and a file sitting in a cloud bucket ready for translation.

System Requirements

- **R** 4.0 or later
- **RStudio** (optional but recommended)
- An internet connection
- A free AutoDesk account (created below)

Step 1: Create an AutoDesk Account

If you do not already have an AutoDesk account, register for free at autodesk.com. A standard account gives you access to the APS free tier, which is sufficient for development and moderate usage.

Step 2: Register an APS Application

APS uses OAuth 2.0 — every API call needs a token, and every token comes from a registered app. Here's how to get one:

1. Go to the APS Developer Portal and sign in.
2. Click **My Apps → Create Application**.
3. Give your app a name (e.g., `AutoDeskR-dev`) and enable the APIs you need. For the examples in this book: **Data Management**, **Model Derivative**, and **Design Automation**.
4. Click **Create**. Your **Client ID** and **Client Secret** appear on the app detail page.

Keep these private. If they end up in a public Git repo, rotate them immediately.

Step 3: Store Credentials Securely

The safest way to use credentials in R is through environment variables stored in `~/.Renviron`. Open or create that file:

```
usethis::edit_r_environ()
```

Add the following lines, replacing the placeholder values:

```
client_id=your_client_id_here
client_secret=your_client_secret_here
```

Save the file and restart R. You can then retrieve the values in any script without hard-coding them:

```
id     <- Sys.getenv("client_id")
secret <- Sys.getenv("client_secret")
```

See the Authentication chapter for a full explanation of OAuth scopes and token management.

Step 4: Install AutoDeskR

Install the stable release from CRAN:

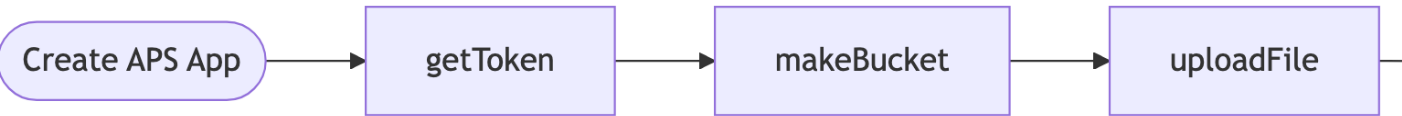
```
install.packages("AutoDeskR")
```

Or install the development version from GitHub:

```
devtools::install_github("paulgovan/AutoDeskR")
```

Step 5: Your First Workflow

The diagram below shows the typical flow through the AutoDeskR APIs:



The following code demonstrates a complete round-trip: authenticate, create a storage bucket, upload a file, and verify the upload.

```
library(AutoDeskR)

# Read credentials from environment
id     <- Sys.getenv("client_id")
secret <- Sys.getenv("client_secret")

# 1. Get an access token with the scopes needed for data management
token <- getToken(id, secret, scope = "data:read data:write bucket:create bucket:read")
token$content$access_token
#> [1] "eyJhbGciOiJSUzI1NiIsImtpZCI6IlU3c1BKNVVpOWNaVjlIR2FhX1pIeDhEd3VYSHcifQ..."
token$content$expires_in
#> [1] 3600

# 2. Create a persistent storage bucket (bucket names must be globally unique)
bucket <- makeBucket(token, bucketKey = "my-first-autodeskr-bucket", policyKey = "persistent")
bucket$content$bucketKey
#> [1] "my-first-autodeskr-bucket"
bucket$content$policyKey
#> [1] "persistent"

# 3. Upload a local file ( 100 MB)
result <- uploadFile(token,
                     localPath = "path/to/your/file.dwg",
                     bucketKey = bucket$bucketKey)
result$content$objectKey
#> [1] "file.dwg"
result$content$size
#> [1] 3145728
```

```r
# 4. List objects in the bucket to confirm the upload succeeded
objects <- listObjects(token, bucketKey = bucket$bucketKey)
print(objects$content$items)
#>   objectKey                                                    objectId    size
#> 1  file.dwg  urn:adsk.objects:os.object:my-first-autodeskr-bucket/file.dwg  3145728
```

If the upload succeeds, `listObjects()` will return a data frame containing your file's `objectKey`, `objectId` (a URN), and size. The `objectId` is used in subsequent chapters when translating the file or loading it in the viewer.

What's Next

With your credentials configured and a file uploaded, you are ready to explore the full API:

Chapter	What it covers
Authentication	Token scopes, token refresh, rate-limit handling
Data Management	Buckets, large-file uploads, pagination
Model Derivative	Translating files to OBJ, STL, SVF; extracting metadata
Design Automation	DWG-to-PDF conversion pipelines
Reality Capture	Photo-to-3D mesh from drone or phone images
Viewer	Embedding the 3D viewer in Shiny and R Markdown
Mesh Reading	Importing OBJ and STL files; inspecting geometry
Mesh Metrics	Surface area, volume, bounding box
3D Visualisation	Interactive rgl widgets and rayshader renders
Point Cloud Analysis	LAS/LAZ files; height normalisation; density rasters
Layer Analysis	Counting and summarising DWG layers
Attribute Extraction	Reading object properties from Model Derivative
Cross-Drawing Comparison	Diffing layer tables across drawing revisions
Drawing Reports	Parameterised gt reports rendered with Quarto
Tandem Overview	Connecting to AutoDesk Tandem facilities
Sensor Streams	Fetching live sensor readings
Live Dashboards	Viewer + sensor chart in a single Shiny app
MCP Server	Exposing AutoDeskR as an AI agent tool
Troubleshooting	Common errors and how to fix them

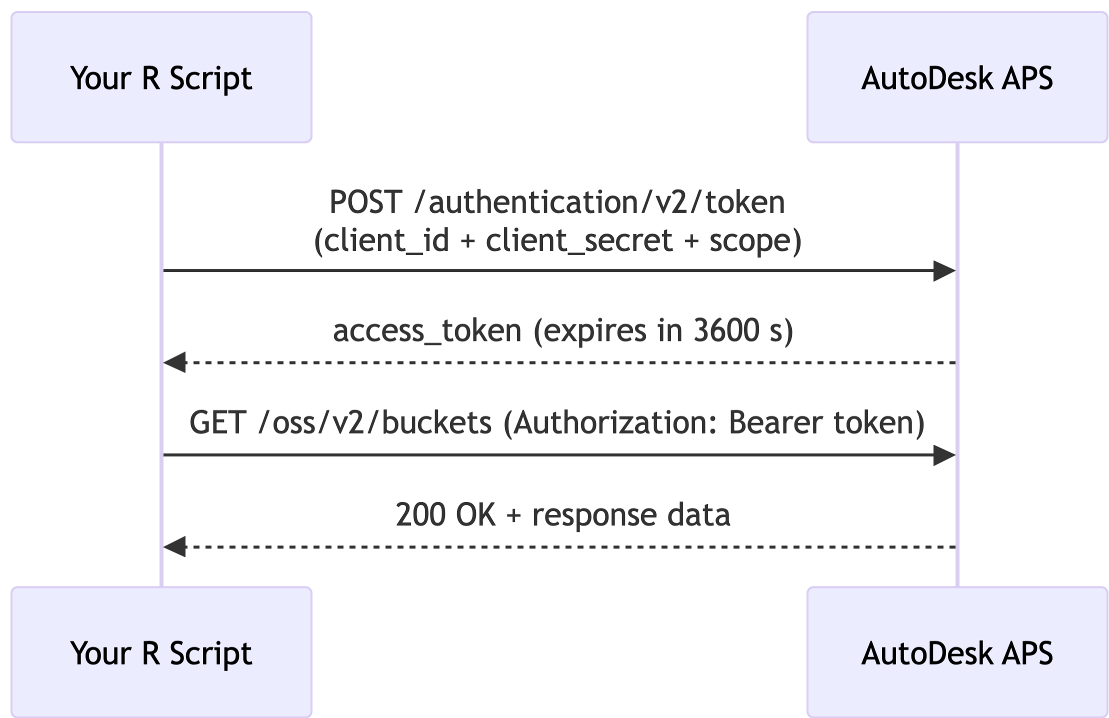
Part III

AutoDesk Platform Services

1 Authentication

Before AutoDeskR can talk to any APS endpoint, it needs a token — a temporary credential that proves your app is who it says it is. This chapter covers how to get one, keep it safe, and deal with the inevitable moment when it expires.

APS uses **2-legged OAuth 2.0** (client credentials flow): your app sends its Client ID and Secret, and AutoDesk hands back an access token. No user login involved, perfect for server-side scripts and automated pipelines.



1.1 Store Credentials Safely

Your Client ID and Secret are like a username and password for your app. Keep them out of your source files and version control. The cleanest approach in R is to stash them in `~/.Renviron`:

```
client_id = "YOUR_CLIENT_ID"
client_secret = "YOUR_CLIENT_SECRET"
token = "YOUR_ACCESS_TOKEN"
urn = "YOUR_URN"
```

Reload the file without restarting R:

```
readRenviron("~/.Renviron")
```

Then pull values at runtime with `Sys.getenv()` — credentials never appear in your code. See Wrapping APIs in the httr2 docs for more on this pattern.

1.2 Get an Access Token

One function call does it all:

```
resp    <- getToken(id = Sys.getenv("client_id"), secret = Sys.getenv("client_secret"))
myToken <- resp$content$access_token
myToken
#> [1] "eyJhbGciOiJSUzI1NiIsImtpZCI6IlU3c1BKNVVpOWNaVjlIR2FhX1pIeDhEd3VYSHcifQ.
#>       eyJjbGllbnRfaWQiOiJBQkNERUYxMjM0NTYiLCJleHAiOjE3NDU0MDA0MDAsInNjb3BlIjpb
#>       ImRhdGE6cmVhZCJdfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c"
```

The full response also tells you what you're working with:

```
str(resp$content)
#> List of 3
#>  $ access_token: chr "eyJhbGciOiJSUzI1NiIsImtpZCI6IlU3c1BKNVVpOWNaVjlIR..."
#>  $ token_type  : chr "Bearer"
#>  $ expires_in  : int 3600
```

> ⚠ Warning
>
> **Tokens expire after 1 hour.** A `401 Unauthorized` mid-session almost always means your token timed out — just call `getToken()` again.
>
> ```
> resp$status_code
> #> [1] 401
> resp$content$errorCode
> #> [1] "AUTH-001"
> resp$content$developerMessage
> #> [1] "The provided access token has expired."
> ```

1.3 Scopes — Only Ask for What You Need

The `scope` parameter controls which APIs the token can access. It's good practice to request the minimum scopes your task actually requires:

Scope	What it unlocks
`data:read`	Read files and derivatives
`data:write`	Upload and modify files
`data:create`	Create new files
`data:search`	Search within files
`bucket:create`	Create storage buckets
`bucket:read`	View bucket details
`bucket:update`	Modify bucket settings
`bucket:delete`	Delete buckets
`code:all`	Design Automation APIs
`account:read`	Read account information
`account:write`	Modify account settings
`user-profile:read`	Read user profile data

Pass multiple scopes as a space-separated string:

```
resp <- getToken(
  id     = Sys.getenv("client_id"),
  secret = Sys.getenv("client_secret"),
  scope  = "bucket:create bucket:read data:write"
)
myToken <- resp$content$access_token
```

1.4 Rate Limits

APS enforces per-application rate limits (typically 60–300 requests per minute depending on the endpoint). Hit the ceiling and you get **HTTP 429**:

```
resp$status_code
#> [1] 429
resp$content$errorCode
#> [1] "TOO-MANY-REQUESTS"
resp$content$developerMessage
#> [1] "Rate limit exceeded. Retry after 60 seconds."
```

This simple retry helper wraps any function call that might get rate-limited:

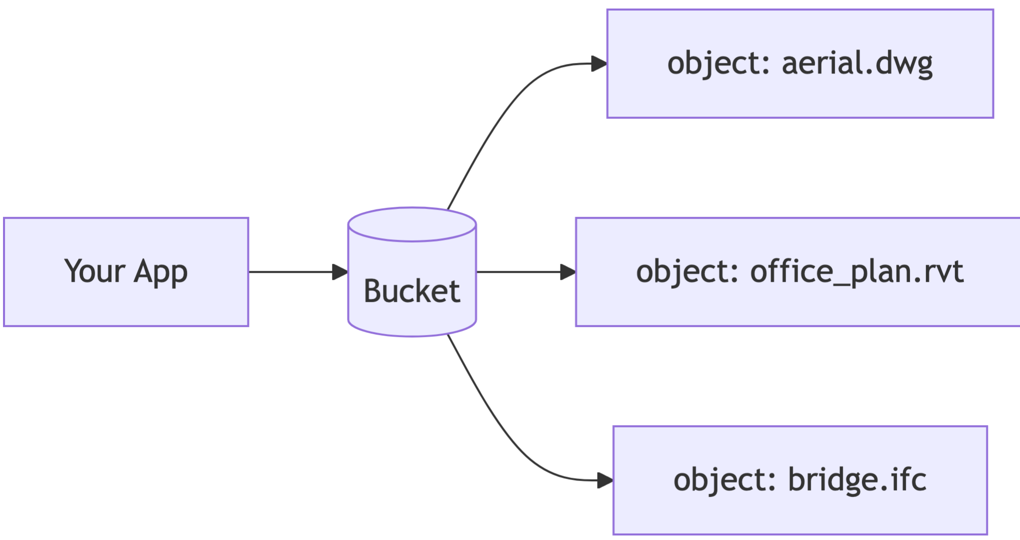
```
aps_retry <- function(fn, max_tries = 3, wait = 10) {
  for (i in seq_len(max_tries)) {
    resp <- fn()
    if (!identical(resp$status_code, 429L)) return(resp)
    message("Rate limited - waiting ", wait, "s (attempt ", i, "/", max_tries, ")")
    Sys.sleep(wait)
  }
```

```
  stop("Max retries exceeded after ", max_tries, " attempts.")
}

# Example: poll translation status with automatic backoff
resp <- aps_retry(function() checkFile(urn = myEncodedUrn, token = myToken))
```

2 Data Management

Think of the Data Management API as AutoDesk's cloud file system. Before you can translate a file, embed it in a viewer, or run Design Automation on it, the file needs to live up in APS storage. This chapter covers everything from creating a bucket to deleting one.

The storage model is simple: your app owns **buckets**, and buckets hold **objects** (your files).



2.1 Create a Bucket

Get a token first - bucket creation needs `bucket:create` and `bucket:read` scopes at minimum:

```
resp <- getToken(
  id     = Sys.getenv("client_id"),
  secret = Sys.getenv("client_secret"),
  scope  = "bucket:create bucket:read data:write"
)
myToken <- resp$content$access_token
#> [1] "eyJhbGciOiJSUzI1NiIsImtpZCI6IlU3c1BKNVVpOWNaVjlIR2FhX1pIeDhEd3VYSHcifQ..."
```

Bucket retention policy controls how long your objects stick around:

Policy	Files live for
`"transient"`	24 hours (great for testing)
`"temporary"`	30 days
`"persistent"`	Until you delete them

> ⚠ Warning
>
> **Bucket names are globally unique** — not just within your app, but across every APS application everywhere. A name like `"test"` is long gone. Use something specific to your organisation and project, e.g. `"acmecorp-bridge-analysis-prod"`. A duplicate name returns a `409 Conflict`:
>
> ```
> resp$status_code
> #> [1] 409
> resp$content$reason
> #> [1] "Bucket already exists"
> ```

```
resp <- makeBucket(token = myToken, bucket = "mybucket", policy = "persistent")
resp$content
#> $bucketKey
#> [1] "mybucket"
#>
#> $bucketOwner
#> [1] "3MVG99OxTyEMCQ3gNp2PjtXbmT8T_MnfCTtpQkuiJNxGtXuoKP7"
#>
#> $createdDate
#> [1] 1745356800000
#>
#> $policyKey
#> [1] "persistent"
```

Check on a bucket anytime with `checkBucket()`:

```
resp <- checkBucket(token = myToken, bucket = "mybucket")
resp$content$policyKey
#> [1] "persistent"
```

2.2 Upload a File

`uploadFile()` handles files up to 100 MB. It returns the object's `objectId`, a URN that every other API will ask for, so save it somewhere handy:

```r
resp  <- uploadFile(
  file   = system.file("samples/aerial.dwg", package = "AutoDeskR"),
  token  = myToken,
  bucket = "mybucket"
)
myUrn <- resp$content$objectId
resp$content
#> $bucketKey
#> [1] "mybucket"
#>
#> $objectId
#> [1] "urn:adsk.objects:os.object:mybucket/aerial.dwg"
#>
#> $objectKey
#> [1] "aerial.dwg"
#>
#> $size
#> [1] 3145728
#>
#> $contentType
#> [1] "application/octet-stream"
#>
#> $location
#> [1] "https://developer.api.autodesk.com/oss/v2/buckets/mybucket/objects/aerial.dwg"
```

2.2.1 Big Files (> 100 MB)

For anything over 100 MB, use `uploadFileSigned()`. iI routes the transfer through signed S3 URLs to sidestep timeout limits:

```r
resp  <- uploadFileSigned(
  file   = "path/to/large_model.rvt",
  token  = myToken,
  bucket = "mybucket"
)
myUrn <- resp$content$objectId
resp$content$size
#> [1] 524288000
```

2.3 List Buckets and Objects

See all buckets your app owns. Paginate with `limit` and `startAt`:

```
resp <- listBuckets(token = myToken, limit = 10)
resp$content$items
#>                          bucketKey    createdDate  policyKey
#> 1                         mybucket  1745356800000 persistent
#> 2                     archive-2025  1740000000000  temporary
#> 3                visualization-dev  1738000000000  transient

# Next page
resp <- listBuckets(token = myToken, limit = 10, startAt = "visualization-dev")
```

Filter by region if your app spans geographies:

```
resp <- listBuckets(token = myToken, region = "EMEA")
resp$content$items
#>       bucketKey    createdDate  policyKey
#> 1  emea-models  1742000000000 persistent
```

List the files inside a bucket:

```
resp <- listObjects(token = myToken, bucket = "mybucket", limit = 10)
resp$content$items
#>          objectKey                                              objectId     size
#> 1       aerial.dwg  urn:adsk.objects:os.object:mybucket/aerial.dwg      3145728
#> 2  office_plan.rvt  urn:adsk.objects:os.object:mybucket/office_plan.rvt 52428800
#> 3       bridge.ifc   urn:adsk.objects:os.object:mybucket/bridge.ifc       8388608
```

2.4 Delete Objects and Buckets

Remove a specific file:

```
resp <- deleteObject(token = myToken, bucket = "mybucket", object = "aerial.dwg")
resp$status_code
#> [1] 200
```

Remove the whole bucket:

```
resp <- deleteBucket(token = myToken, bucket = "mybucket")
resp$status_code
#> [1] 200
```

Warning

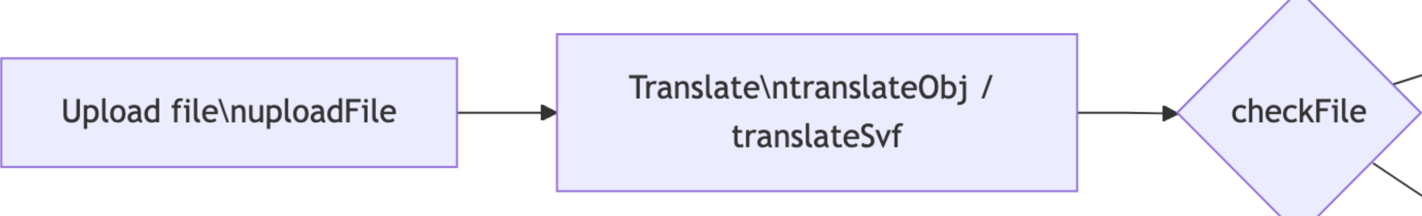
Empty the bucket first. `deleteBucket()` won't touch a bucket that still has objects in it. Delete everything with `deleteObject()` first, or you'll get:

```
resp$status_code
#> [1] 409
resp$content$reason
#> [1] "Bucket not empty"
```

3 Model Derivative

A DWG file sitting in a bucket is just bytes. The Model Derivative API is what turns those bytes into something useful, OBJ and STL meshes you can analyse in R, SVF viewables you can embed in a browser, or structured metadata you can query with `dplyr`. This chapter covers the full translation workflow.

The pattern is always the same: submit a job, poll until it finishes, then pull the output.



3.1 Translate to OBJ

First get a token with `data:read` and `data:write` scopes. Note that only certain source formats can produce OBJ output (DWG, IPT, IAM, IPN, F3D, FBX. See the APS supported formats list for the full matrix).

```
resp <- getToken(
  id     = Sys.getenv("client_id"),
  secret = Sys.getenv("client_secret"),
  scope  = "data:read data:write"
)
myToken <- resp$content$access_token
```

APS requires the file's URN to be Base-64 encoded before it'll accept it — `jsonlite::base64_enc()` handles that:

```
myEncodedUrn <- jsonlite::base64_enc(myUrn)
myEncodedUrn
#> [1] "dXJuOmFkc2sub2JqZWN0czpvcy5vYmplY3Q6bXlidWNrZXQvYWVyaWFsLmR3Zw=="
```

Kick off the translation:

```
resp <- translateObj(urn = myEncodedUrn, token = myToken)
resp$content
#> $result
#> [1] "created"
#>
#> $urn
#> [1] "dXJuOmFkc2sub2JqZWN0czpvcy5vYmplY3Q6bXlidWNrZXQvYWVyaWFsLmR3Zw=="
#>
#> $acceptedJobs$output$formats[[1]]$type
#> [1] "obj"
```

> ⚠ Warning
>
> **OBJ translation only accepts certain source formats.** Submitting an unsupported format returns a `415 Unsupported Media Type`:
>
> ```
> resp$status_code
> #> [1] 415
> resp$content$diagnostic
> #> [1] "File type not supported for OBJ translation."
> ```

Poll `checkFile()` until `status` hits `"success"`. The job can take anywhere from a few seconds to a few minutes depending on file complexity:

```
resp <- checkFile(urn = myEncodedUrn, token = myToken)
resp$content
#> $status
#> [1] "success"
#>
#> $progress
#> [1] "complete"
#>
#> $region
#> [1] "US"
```

> ℹ Note
>
> Translation status moves through `"pending"` → `"inprogress"` → `"success"`. Don't try to download before you see `"success"` — there's nothing there yet. Use the `aps_retry()` helper from the Authentication chapter to handle rate limiting while you wait.

Retrieve the output URN, then download:

```r
resp <- getOutputUrn(urn = myUrn, token = Sys.getenv("token"))
myOutputUrn        <- resp$content$derivatives[[1]]$children[[1]]$urn
myEncodedOutputUrn <- jsonlite::base64_enc(myOutputUrn)

resp <- downloadFile(
  urn        = myEncodedUrn,
  output_urn = myEncodedOutputUrn,
  token      = myToken,
  destfile   = "aerial.obj"
)
resp$status_code
#> [1] 200
```

3.2 Translate to STL

Same pattern as OBJ, different function:

```r
resp <- translateStl(urn = myEncodedUrn, token = myToken)
resp$content$acceptedJobs$output$formats[[1]]$type
#> [1] "stl"

# Poll until done
resp <- checkFile(urn = myEncodedUrn, token = myToken)
resp$content$status
#> [1] "success"
```

3.3 Extract Metadata from a File

SVF format unlocks the structured metadata inside a model. The object tree, property sets, layer information. This is the format the Viewer also uses.

```r
resp <- translateSvf(urn = myEncodedUrn, token = myToken)
resp$content$result
#> [1] "created"

# Wait for it...
resp <- checkFile(urn = myEncodedUrn, token = myToken)
resp$content$status
#> [1] "success"
```

`getMetadata()` returns the viewable GUIDs. You'll need the GUID for every subsequent metadata call:

```r
resp   <- getMetadata(urn = myEncodedUrn, token = myToken)
myGuid <- resp$content$data$metadata[[1]]$guid
resp$content$data
#> $type
#> [1] "metadata"
#>
#> $metadata[[1]]$name
#> [1] "aerial"
#>
#> $metadata[[1]]$guid
#> [1] "a7b3c9d2-4e1f-4a8b-b6c2-9d3e7f5a1b4c"
#>
#> $metadata[[1]]$isMasterView
#> [1] TRUE
```

`getObjectTree()` returns the model's full object hierarchy. For a DWG, the children of the root node are its layers:

```r
resp <- getObjectTree(guid = myGuid, urn = myEncodedUrn, token = myToken)
resp$content$data$objects[[1]]
#> $objectid
#> [1] 1
#>
#> $name
#> [1] "aerial.dwg"
#>
#> $objects[[1]]$name
#> [1] "Layer: A-SITE"
#>
#> $objects[[2]]$name
#> [1] "Layer: A-BLDG"
#>
#> $objects[[3]]$name
#> [1] "Layer: A-ROAD"
```

`getData()` pulls all geometry properties and material data, bounding boxes, layer assignments, custom attributes. This is the raw material for the Layer Structure Analysis chapter:

```r
resp <- getData(guid = myGuid, urn = myEncodedUrn, token = myToken)
resp$content$data$collection[[1]]
#> $objectid
#> [1] 2
#>
#> $name
#> [1] "Layer: A-SITE"
#>
```

```
#> $properties$`Layer and Material`$Layer
#> [1] "A-SITE"
#>
#> $properties$Geometry$`Bounding Box Min X`
#> [1] -142.83
#>
#> $properties$Geometry$`Bounding Box Max X`
#> [1]  318.62
```

> ⚠ Warning
>
> **`getData()` can be large.** Complex models return thousands of objects in a single response. For very large DWGs, consider filtering the object tree first to identify only the layers or elements you care about.

4 Design Automation

Need to convert a DWG to PDF without spinning up an AutoCAD licence? Design Automation is AutoDesk's answer to cloud-based CAD scripting. You submit a work item, AutoDesk runs it on their infrastructure, and the output lands in a destination URL you specify.

This chapter focuses on the most common use case: DWG → PDF.

4.1 Convert a DWG to PDF

`makePdf()` submits the conversion job. The token needs the `code:all` scope. This is specific to Design Automation and different from the data scopes used elsewhere.

Warning

Both source and destination must be publicly accessible URLs. Private Google Drive links, localhost paths, and short-lived signed S3 URLs won't work. The AutoDesk cloud workers need to be able to reach them directly. Use a public host or a pre-signed URL with plenty of time left on it.

```
resp <- getToken(
  id     = Sys.getenv("client_id"),
  secret = Sys.getenv("client_secret"),
  scope  = "code:all"
)
myToken <- resp$content$access_token
```

```
mySource      <- "http://download.autodesk.com/us/samplefiles/acad/visualization_-_aerial.dwg"
myDestination <- "https://drive.google.com/folderview?id=0BygncDVHf60mTDZVNDltLThLNmM&usp=shar

resp         <- makePdf(source = mySource, destination = myDestination, token = myToken)
myWorkItemId <- resp$content$id
resp$content
#> $id
#> [1] "f47ac10b-58cc-4372-a567-0e02b2c3d479"
#>
#> $status
#> [1] "pending"
#>
#> $stats$timeQueued
#> [1] "2026-04-23T10:15:30.412Z"
```

4.2 Poll for Completion

Use the work item `id` to check status. A simple DWG typically takes **15–30 seconds**; complex drawings with many layers or external references take longer.

```
repeat {
  resp <- checkPdf(id = myWorkItemId, token = myToken)
  cat("Status:", resp$content$status, "\n")
  if (resp$content$status == "success") break
  Sys.sleep(5)
}
#> Status: pending
#> Status: inprogress
#> Status: success
```

The full timing breakdown when it's done:

```
resp$content
#> $id
#> [1] "f47ac10b-58cc-4372-a567-0e02b2c3d479"
#>
#> $status
#> [1] "success"
#>
#> $stats$timeQueued
#> [1] "2026-04-23T10:15:30.412Z"
#>
#> $stats$timeDownloadStarted
#> [1] "2026-04-23T10:15:31.804Z"
#>
```

```
#> $stats$timeInstructionsStarted
#> [1] "2026-04-23T10:15:34.217Z"
#>
#> $stats$timeFinished
#> [1] "2026-04-23T10:15:48.993Z"
```

Once `status` is `"success"`, the PDF is waiting at your `destination` URL.

4.3 Batch Conversion

Converting a folder of DWGs is just a loop:

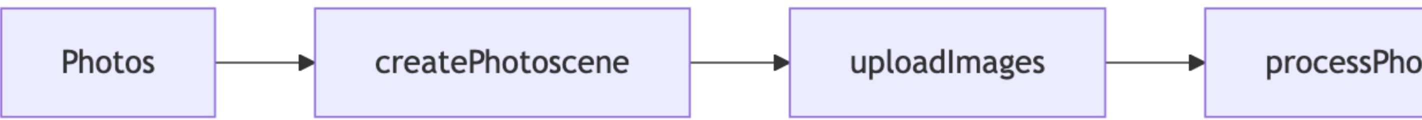
```r
dwg_urls  <- c(
  "https://files.example.com/floor_plan.dwg",
  "https://files.example.com/site_plan.dwg",
  "https://files.example.com/elevations.dwg"
)
dest_base <- "https://output.example.com/pdfs/"

job_ids <- vapply(dwg_urls, function(url) {
  fname <- sub(".*/", "", url)
  resp  <- makePdf(source      = url,
                   destination = paste0(dest_base, sub("\\.dwg$", ".pdf", fname)),
                   token       = myToken)
  resp$content$id
}, character(1))

# Poll until all jobs complete
repeat {
  statuses <- vapply(job_ids, function(id) {
    checkPdf(id = id, token = myToken)$content$status
  }, character(1))
  cat(format(Sys.time(), "%H:%M:%S"), "-", paste(statuses, collapse = " | "), "\n")
  if (all(statuses == "success")) break
  Sys.sleep(10)
}
#> 10:20:05 - pending | pending | inprogress
#> 10:20:15 - success | inprogress | inprogress
#> 10:20:25 - success | success | success
```

5 Reality Capture

Got a folder of overlapping photos and want a 3D mesh? That's the Reality Capture API (also called Photo-to-3D). You upload the images, AutoDesk's photogrammetry engine does the heavy lifting, and you download a georeferenced 3D model — no specialised hardware or software required.

Common uses: - **As-built documentation** — fly a drone over a construction site each week and generate a mesh for each visit - **Heritage recording** — photograph a structure with a phone and get a measurable 3D record - **Site topography** — turn aerial photos into a terrain mesh for civil design - **Mesh analysis** — feed the output into `Rvcg` via the 3D Geometry chapters

The full workflow goes like this:



5.1 Before You Start

Tip

Good photos make good meshes. A few guidelines that will save you a lot of frustration:

- At least **20 images** per scene — 50–200 is typical for a small structure
- **60–80 % overlap** between adjacent shots; every surface should appear in at least 3 photos
- Consistent, diffuse lighting — harsh shadows and mirror-like reflections confuse the algorithm

- Move the camera *around* the subject, don't just zoom in from one spot
- GPS/EXIF tags are optional but improve georeferencing accuracy

5.2 Step 1: Authenticate

Reality Capture needs `data:read` and `data:write` scopes:

```
library(AutoDeskR)

resp    <- getToken(
  id     = Sys.getenv("client_id"),
  secret = Sys.getenv("client_secret"),
  scope  = "data:read data:write"
)
myToken <- resp$content$access_token
```

5.3 Step 2: Create a Photoscene

A **photoscene** is the container for your images and output. The `format` parameter controls what comes out the other end:

Format	Output
`"rcm"`	AutoDesk ReCap project (default)
`"rcs"`	ReCap point cloud — feeds into Point Cloud Analysis
`"obj"`	Wavefront OBJ mesh — feeds into Reading OBJ and STL Meshes
`"ortho"`	Orthophoto + DSM GeoTIFF
`"report"`	PDF quality report

```
ps             <- createPhotoscene(name = "site-survey-2026", format = "obj", token = myToken)
myPhotosceneId <- ps$content$photoscene$photosceneid
myPhotosceneId
#> [1] "K2RDAPFINOOPKGU9B65Q2TJD"
```

5.4 Step 3: Upload Images

Pass a character vector of local file paths. AutoDeskR bundles them into a single multipart POST. JPEG and PNG both work.

```r
image_files <- list.files("photos/", pattern = "\\.jpg$", full.names = TRUE)

imgs <- uploadImages(
  photoscene_id = myPhotosceneId,
  files         = image_files,
  token         = myToken
)
imgs$content$Files$file[[1]]
#> $filename
#> [1] "site_001.jpg"
#>
#> $fileid
#> [1] "A1B2C3D4E5F6G7H8"
#>
#> $filesize
#> [1] 4718592
```

> ⚠ Warning
>
> **Free-tier limits:** Maximum **1,000 images per photoscene** and **10 photoscenes per month**. Exceeding the quota returns a `403 Forbidden`. Upgrade to a paid plan for higher limits.

5.5 Step 4: Start Processing

One call kicks off the reconstruction:

```r
proc <- processPhotoscene(photoscene_id = myPhotosceneId, token = myToken)
proc$content$photoscene$progressmsg
#> [1] "Queued"
```

5.6 Step 5: Wait for It

5.6.1 Option A — Block until done (recommended)

`waitForPhotoscene()` polls on a schedule and returns when the job finishes. Set `verbose = TRUE` to watch the progress tick up:

```r
done <- waitForPhotoscene(
  photoscene_id = myPhotosceneId,
  token         = myToken,
  interval      = 60,    # check every 60 seconds
  timeout       = 3600,  # give up after an hour
```

```
  verbose      = TRUE
)
#> Reality Capture progress: 0 - Queued
#> Reality Capture progress: 10 - Uploading files
#> Reality Capture progress: 35 - Calibrating cameras
#> Reality Capture progress: 60 - Building point cloud
#> Reality Capture progress: 85 - Generating mesh
#> Reality Capture progress: 100 - Done
```

5.6.2 Option B — Manual polling

If you want more control, call `checkPhotoscene()` yourself:

```
status <- checkPhotoscene(photoscene_id = myPhotosceneId, token = myToken)
status$content$photoscene$progress
#> [1] "75"
status$content$photoscene$progressmsg
#> [1] "Processing geometry"
```

> ⚠ Warning
>
> **Processing time scales with image count.** A 50-image scene typically takes 5–15 minutes; 500 images may take over an hour. Set `timeout` generously — there's nothing worse than a premature timeout on an 80-minute job.

5.7 Step 6: Download the Output

When `progress` hits `"100"`, retrieve the download URL and save the mesh locally:

```
output_url <- done$content$photoscene$scenelink

download.file(
  url      = paste0(output_url, "/output.obj"),
  destfile = "site-survey-2026.obj",
  headers  = c(Authorization = paste("Bearer", myToken))
)
```

From here, the OBJ file is ready for Reading OBJ and STL Meshes or further translation via the Model Derivative API.

6 Viewer

The AutoDesk Viewer is a WebGL-based 3D model browser that runs entirely in the browser. With AutoDeskR, you can embed it in a Shiny app with a single function call — no JavaScript required. Rotate, zoom, and inspect your models without leaving R.

Before launching the viewer, you need a file translated to SVF format and its encoded URN — the Model Derivative chapter covers both steps.

6.1 Launch the Viewer

`viewer3D()` opens a Shiny app with the viewer pre-loaded. The `viewerType` parameter controls the UI chrome:

`viewerType`	What you get
`"header"`	Full toolbar, navigation panels, and settings (default)
`"headless"`	Viewer canvas only — clean embed for dashboards
`"vr"`	WebVR mode, optimised for mobile

```
resp <- getToken(
  id     = Sys.getenv("client_id"),
  secret = Sys.getenv("client_secret"),
  scope  = "data:read"
)
myToken      <- resp$content$access_token
myEncodedUrn <- jsonlite::base64_enc(Sys.getenv("urn"))

# Full viewer - great for exploration
viewer3D(urn = myEncodedUrn, token = myToken)

# Headless - embed in a dashboard without the toolbar chrome
viewer3D(urn = myEncodedUrn, token = myToken, viewerType = "headless")

# VR mode for a phone or headset
viewer3D(urn = myEncodedUrn, token = myToken, viewerType = "vr")
```

> **Note**
>
> When you run `viewer3D()` in RStudio or a browser, a WebGL viewport opens and you can rotate, zoom, and click elements to inspect their properties. The viewer renders whatever SVF translation produced: layer geometry for DWG files, full BIM elements for Revit.

6.2 Embed in a Shiny App

`viewerUI()` drops the viewer into any Shiny layout as a module. The `id` parameter namespaces the module so multiple viewers can live on the same page without stepping on each other.

```
library(shiny)
library(AutoDeskR)

ui <- fluidPage(
  viewerUI("viewer1", urn = myEncodedUrn, token = myToken)
)

server <- function(input, output, session) {}

shinyApp(ui, server)
```

6.2.1 Viewer in a Tab Panel

Headless mode is perfect for a multi-tab layout — the viewer sits in one tab and data tables or charts in another:

```
ui <- fluidPage(
  tabsetPanel(
    tabPanel("Model",
      viewerUI("viewer1", myEncodedUrn, myToken, viewerType = "headless")
    ),
    tabPanel("Metadata",
      verbatimTextOutput("meta")
    ),
    tabPanel("Layer Chart",
      plotOutput("layer_plot")
    )
  )
)

server <- function(input, output, session) {
  output$meta <- renderPrint({
    # getData() results displayed alongside the viewer
```

```
  })
}

shinyApp(ui, server)
```

6.2.2 Viewer + Sensor Dashboard

Pair the headless viewer with a `dygraphs` time-series panel for a live digital twin layout — see the Live Dashboards chapter for the full implementation:

```
library(dygraphs)

ui <- fluidPage(
  fluidRow(
    column(7, viewerUI("twin_viewer", myEncodedUrn, myToken, viewerType = "headless")),
    column(5, dygraphOutput("sensor_plot", height = "500px"))
  )
)
```

6.3 What's Displayed

The viewer renders whatever the SVF translation produced. For a DWG file, that's the full drawing geometry organised by layer. For a Revit file, it's the full BIM model with element properties accessible in the side panel. Click any element in the viewer to inspect its properties — the same data that `getData()` returns programmatically.

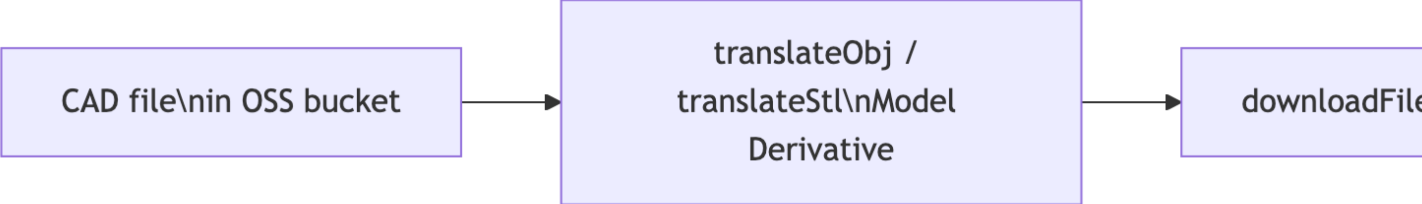
Part IV

3D Geometry & Meshes

7 Reading OBJ and STL Meshes

Once you've translated a CAD file through the Model Derivative API, you've got an OBJ or STL file sitting on disk. Now the fun part: loading it into R.

`rgl` (Adler, Murdoch, et al. 2024) and `Rvcg` (Schlager 2024) both read OBJ and STL files. `rgl` is great for interactive viewing; `Rvcg` is the one to reach for when you want to compute things. It returns a `tmesh3d` object that plays nicely with the whole `Rvcg` family of analysis functions.



7.1 Prerequisites

```r
install.packages(c("rgl", "Rvcg"))
```

7.2 Getting the OBJ File

The full upstream chain, translate, poll, get output URN, download, picks up where the Model Derivative chapter left off:

```r
library(AutoDeskR)

resp     <- getToken(id     = Sys.getenv("client_id"),
                     secret = Sys.getenv("client_secret"),
                     scope  = "data:read data:write")
myToken  <- resp$content$access_token

myEncodedUrn <- jsonlite::base64_enc(Sys.getenv("urn"))

translateObj(urn = myEncodedUrn, token = myToken)

repeat {
  status <- checkFile(urn = myEncodedUrn, token = myToken)
```

```r
  if (status$content$status == "success") break
  Sys.sleep(5)
}

outputUrn       <- getOutputUrn(urn = myEncodedUrn, token = myToken)
myOutputUrn     <- outputUrn$content$derivatives[[1]]$children[[1]]$urn
myEncodedOutput <- jsonlite::base64_enc(myOutputUrn)

downloadFile(urn        = myEncodedUrn,
             output_urn = myEncodedOutput,
             token      = myToken,
             destfile   = "aerial.obj")
```

7.3 Reading with Rvcg (recommended for analysis)

`vcgImport()` parses the OBJ and returns a `tmesh3d`, a list with a vertex matrix (`vb`) and a triangle index matrix (`it`). Every analysis function in the next three chapters works on this object.

```r
library(Rvcg)
mesh_vcg <- vcgImport("aerial.obj")
str(mesh_vcg)
#> List of 4
#>  $ vb     : num [1:4, 1:2847] ...   # homogeneous vertex coords (X/Y/Z/W)
#>  $ it     : int [1:3, 1:1832] ...   # triangle face indices
#>  $ normals: num [1:4, 1:2847] ...
#>  $ material: list()
```

7.4 Reading with rgl (for interactive viewing)

`readOBJ()` returns a similar `mesh3d` object that you can render immediately:

```r
library(rgl)
mesh_obj <- readOBJ("aerial.obj")
```

> **Tip**
>
> OBJ files from the Model Derivative API sometimes contain multiple named objects (one per CAD layer or body). `vcgImport()` merges them into a single mesh by default. Use `rgl::readOBJ(material = TRUE)` if you need to keep them separate for per-layer analysis.

7.5 Reading STL

Exactly the same call works for STL files. `vcgImport()` handles both formats transparently:

```
mesh_stl <- vcgImport("aerial.stl")
# Same tmesh3d structure - all subsequent chapters work identically on OBJ and STL
```

7.6 Quick Inspection

A few fast checks to make sure the mesh loaded correctly:

```
cat("Vertices:", ncol(mesh_vcg$vb), "\n")
#> Vertices: 2847
cat("Faces:   ", ncol(mesh_vcg$it), "\n")
#> Faces:    1832

# Coordinate ranges - useful sanity check that units are what you expect
apply(t(mesh_vcg$vb[1:3, ]), 2, range)
#>           [,1]      [,2]   [,3]
#> [1,] -142.830  -98.470  0.000
#> [2,]  318.620  241.190 12.500
```

The `mesh_vcg` object lives on from here through the Mesh Metrics, 3D Visualisation, and Mesh Comparison chapters, so keep it in your environment.

8 Mesh Metrics

Got a mesh? Let's measure it. `Rvcg` (Schlager 2024) and `geometry` (Roussel, Sterratt, et al. 2024) turn a `tmesh3d` into a full geometric report, surface area, volume, bounding box, centroid. These numbers are useful for QA (does the translated geometry match the design intent?), quantity takeoff, and as inputs to dashboards and reports.

This chapter uses `mesh_vcg` from Reading OBJ and STL Meshes.

8.1 Surface Area

```
library(Rvcg)
area <- vcgArea(mesh_vcg)
area
#> [1] 14823.6   # square units - same as the DWG drawing units
```

8.2 Volume

```
vol <- vcgVolume(mesh_vcg)
vol
#> [1] 48219.3
```

Warning

`vcgVolume()` only gives meaningful results for **watertight (closed) meshes**. OBJ files translated from open 2D DWG drawings are almost never watertight, so this number will be unreliable. Use the convex hull fallback below instead.

8.3 Bounding Box

```
bb <- apply(t(mesh_vcg$vb[1:3, ]), 2, range)
rownames(bb) <- c("min", "max")
colnames(bb) <- c("X", "Y", "Z")
bb
```

```
#>          X        Y      Z
#> min -142.83  -98.47   0.00
#> max  318.62  241.19  12.50

diff(bb)   # width / depth / height
#>      X      Y     Z
#> 461.45 339.66 12.50
```

8.4 Centroid

```
centroid <- colMeans(t(mesh_vcg$vb[1:3, ]))
names(centroid) <- c("X", "Y", "Z")
centroid
#>     X      Y     Z
#>  87.90  71.36  6.25
```

8.5 Convex Hull Volume

For open (non-watertight) meshes, `geometry::convhulln()` (Roussel, Sterratt, et al. 2024) computes the volume and surface area of the convex hull, a conservative upper bound that doesn't require a closed surface:

```
library(geometry)
pts  <- t(mesh_vcg$vb[1:3, ])
hull <- convhulln(pts, options = "FA")
hull$vol
#> [1] 1871423.5
hull$area
#> [1] 216834.2
```

8.6 Summary Table

Pull all the numbers together into a single data frame, ready to feed into a `gt` table or write to CSV:

```
mesh_summary <- data.frame(
  metric = c("Vertices", "Faces", "Surface area",
             "Convex hull volume", "Convex hull area",
             "Bounding box X", "Bounding box Y", "Bounding box Z",
             "Centroid X", "Centroid Y", "Centroid Z"),
  value  = c(ncol(mesh_vcg$vb), ncol(mesh_vcg$it), area,
             hull$vol, hull$area,
```

```
             diff(bb)[1], diff(bb)[2], diff(bb)[3],
             centroid[1], centroid[2], centroid[3])
)
mesh_summary
#>                 metric       value
#> 1             Vertices    2847.000
#> 2                Faces    1832.000
#> 3         Surface area   14823.600
#> 4   Convex hull volume 1871423.500
#> 5     Convex hull area  216834.200
#> 6       Bounding box X     461.450
#> 7       Bounding box Y     339.660
#> 8       Bounding box Z      12.500
#> 9           Centroid X      87.900
#> 10          Centroid Y      71.360
#> 11          Centroid Z       6.250
```

8.7 Visualising the Metrics

A quick bar chart makes the bounding box dimensions easy to compare at a glance:

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```
ggplot(bbox_df, aes(x = dimension, y = value, fill = dimension)) +
  geom_col(show.legend = FALSE) +
  geom_text(aes(label = round(value, 1)), vjust = -0.4, size = 3.5) +
  scale_fill_manual(values = c("#367ABF", "#4CAF50", "#F4A261")) +
  labs(title = "Bounding Box Dimensions",
       subtitle = "aerial.dwg - translated mesh",
       x = NULL, y = "Drawing units") +
  theme_minimal()
```

Bounding Box Dimensions

aerial.dwg <U+2014> translated mesh

Drawing units

400
300
200
100
0

339.7
12.5
461.4

Depth (Y)
Height (Z)
Width (X)

These metrics can be passed to a `gt` table for a polished handover report. See Automated Drawing Reports for the formatting details.

9 3D Visualisation

Numbers are useful; actually *seeing* the mesh is better. This chapter covers two routes: `rgl` (Adler, Murdoch, et al. 2024) for interactive exploration in RStudio (spin it with the mouse, zoom in, inspect details), and `rayshader` (Morgan-Wall 2024) for rendering publication-quality static images.

Both work on `mesh_vcg` from Reading OBJ and STL Meshes.

9.1 Interactive 3D with rgl

`shade3d()` opens an OpenGL window where you can rotate and zoom with the mouse. Takes one line:

```
library(rgl)
open3d()
shade3d(mesh_vcg, col = "#C0C0C0", alpha = 0.9)
bg3d("white")
axes3d()
title3d("Aerial DWG - Translated Mesh")
```

9.1.1 Embedding in HTML Output

`rgl` windows are interactive in RStudio but don't embed in HTML documents on their own. `rglwidget()` wraps the scene in a self-contained WebGL widget that works in any browser:

```
# Add this once at the top of your document (setup chunk)
knitr::knit_hooks$set(webgl = rgl::hook_webgl)

open3d()
shade3d(mesh_vcg, col = "#4A90D9", alpha = 0.85)
rglwidget()
```

> **Note**
>
> The WebGL widget is fully self-contained in the rendered HTML. Readers can rotate and zoom without a running R server. It does require a modern browser (Chrome, Firefox, Safari, Edge all work fine).

9.2 Colouring by Height

Map vertex Z-values to a colour ramp to reveal the model's vertical structure instantly. Blues for low, reds for high:

```
z_vals <- mesh_vcg$vb[3, ]
z_norm <- (z_vals - min(z_vals)) / diff(range(z_vals))   # 0-1

cols      <- colorRampPalette(c("#2166ac", "#92c5de", "#f7f7f7",
                                 "#f4a582", "#d6604d"))(100)
vert_cols <- cols[ceiling(z_norm * 99) + 1]

open3d()
shade3d(mesh_vcg, col = vert_cols)
rglwidget()
```

9.3 Rendered Images with rayshader

`rayshader` renders the active `rgl` scene with ray-traced lighting and ambient occlusion, great for reports and presentations. The output is a PNG file.

```
library(rayshader)

vertices  <- t(mesh_vcg$vb[1:3, ])
triangles <- t(mesh_vcg$it)

open3d()
triangles3d(vertices[triangles[, 1], ],
            vertices[triangles[, 2], ],
            vertices[triangles[, 3], ],
            col = "steelblue")

render_snapshot(filename    = "aerial_render.png",
                width       = 1200,
                height      = 900,
                title_text  = "Aerial DWG - Rayshader Render",
                title_color = "white",
                title_size  = 24,
                clear       = TRUE)
```

Include the render in your Quarto document with a standard image link:

```
![Aerial DWG rendered with rayshader](aerial_render.png)
```

9.4 Depth of Field

Add a depth-of-field effect to emphasise the model's 3D structure in the output image, works especially well for tall structures:

```
render_depth(focus       = 0.7,
             focallength = 200,
             filename    = "aerial_dof.png")
```

9.5 Quick Comparison: rgl vs. rayshader

	`rgl`	`rayshader`
Output	Interactive HTML widget	Static PNG
Use for	Exploration, dashboards	Reports, presentations
Render time	Instant	Seconds–minutes
WebGL support	Yes (via `rglwidget()`)	No

10 Point Cloud Analysis

The Reality Capture API can output a point cloud alongside (or instead of) a mesh. Once you've converted it to LAS format, `lidR` (Roussel et al. 2020) takes over — height statistics, density maps, canopy models, voxel volumes. This is the go-to tool for site survey analysis.

10.1 Getting the Point Cloud

Request `rcs` format when creating your photoscene:

```
library(AutoDeskR)

resp    <- getToken(id     = Sys.getenv("client_id"),
                    secret = Sys.getenv("client_secret"),
                    scope  = "data:read data:write")
myToken <- resp$content$access_token

ps <- createPhotoscene(name = "site-2026-q2", token = myToken)
# ... uploadImages(), processPhotoscene(), waitForPhotoscene() ...

result  <- checkPhotoscene(photoscene_id = myPhotosceneId, token = myToken)
rcs_url <- result$content$photoscene$scenelink

download.file(url      = rcs_url,
              destfile = "site-survey.rcs",
              headers  = c(Authorization = paste("Bearer", myToken)))
```

> **Warning**
>
> **`lidR` reads LAS/LAZ, not RCS.** Convert the downloaded `.rcs` file to LAS using CloudCompare (free and open source) before proceeding. In CloudCompare: *File* → *Open* the RCS file, then *File* → *Save As* → LAS format.

10.2 Reading the Point Cloud

```r
library(lidR)
pc <- readLAS("site-survey.las")
pc
#> class        : LAS (v1.3 format 1)
#> memory       : 247.8 Mb
#> extent       : 406325.4, 406412.8, 5765842.6, 5765901.3
#> coord. ref.  : WGS 84 / UTM zone 32N
#> area         : 4064.9 units²
#> points       : 3.62 million pts (1 return)
```

10.3 Height Statistics

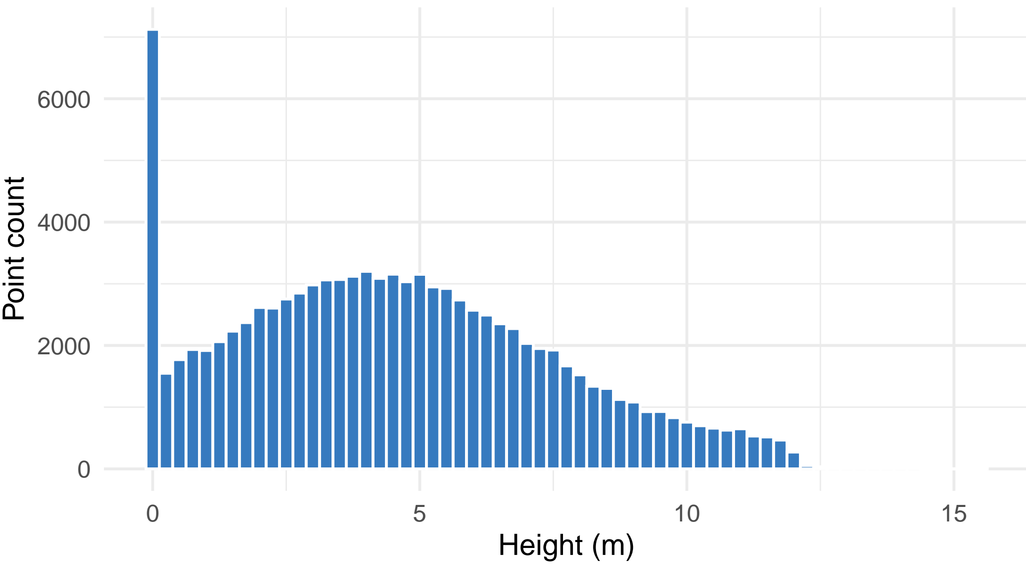
Normalise to ground level before computing anything height-related:

```r
pc_norm <- normalize_height(pc, knnidw())

mean(pc_norm$Z)
#> [1] 4.12    # mean height above ground (m)
sd(pc_norm$Z)
#> [1] 2.87

quantile(pc_norm$Z, probs = c(0.25, 0.5, 0.75, 0.95))
#>   25%   50%   75%   95%
#>  1.20  3.44  6.11 11.83
```

10.4 Height Distribution

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```r
ggplot(data.frame(z = z_sample), aes(x = z)) +
  geom_histogram(binwidth = 0.25, fill = "#367ABF", colour = "white") +
  labs(title = "Point Cloud Height Distribution",
       subtitle = "Normalised above ground level",
       x = "Height (m)", y = "Point count") +
  theme_minimal()
```



10.5 Canopy Height Model

Rasterise the maximum Z per cell to produce a canopy height model — great for visualising roof heights or vegetation structure across the site:

```
chm <- rasterize_canopy(pc_norm, res = 0.5, algorithm = p2r())
plot(chm, col = height.colors(50), main = "Canopy Height Model (0.5 m resolution)")
```

10.6 Point Density Grid

```
density <- grid_density(pc, res = 1)
plot(density, col = gray.colors(50), main = "Point Density (pts/m²)")

mean(density[], na.rm = TRUE)
#> [1] 891.3   # points per square metre
```

10.7 Voxel Volume

```
vox        <- voxelize_points(pc_norm, res = 0.25)
volume_m3  <- nrow(vox) * 0.25^3
cat("Estimated volume:", round(volume_m3, 1), "m³\n")
#> Estimated volume: 1483.2 m³
```

10.8 Summary Table

```
library(ggplot2)

metrics <- data.frame(
  metric = c("Total points", "Survey area (m²)", "Mean density (pts/m²)",
             "Mean height (m)", "95th-pct height (m)", "Voxel volume (m³)"),
  value  = c(npoints(pc), area(pc), mean(density[], na.rm = TRUE),
             mean(pc_norm$Z), quantile(pc_norm$Z, 0.95), volume_m3)
)

ggplot(metrics, aes(x = reorder(metric, value), y = value)) +
  geom_col(fill = "#367ABF") +
  coord_flip() +
  labs(title = "Point Cloud Summary Metrics",
       x = NULL, y = "Value") +
  theme_minimal()
```

11 Mesh Comparison

As-designed vs. as-built: did the construction actually match the model? `Rvcg` (Schlager 2024) lets you quantify that gap by computing per-vertex surface distances between two meshes, and colour the design mesh by how far each point drifted.

The typical setup: the design mesh comes from a DWG translated via the Model Derivative API; the as-built mesh comes from Reality Capture photogrammetry.

11.1 Loading Two Meshes

```
library(Rvcg)

mesh_design <- vcgImport("aerial_design.obj")   # as-designed
mesh_built  <- vcgImport("aerial_asbuilt.obj")  # from Reality Capture

cat("Design vertices:  ", ncol(mesh_design$vb), "\n")
#> Design vertices:   2847
cat("As-built vertices:", ncol(mesh_built$vb),  "\n")
#> As-built vertices: 38124
```

11.2 Smoothing the As-Built Mesh

Photogrammetric meshes carry surface noise from image reconstruction. A few iterations of Laplacian smoothing tames that without destroying the coarse geometry:

```
mesh_built_smooth <- vcgSmooth(mesh_built, iteration = 3, lambda = 0.5)
```

11.3 Surface Distance

`vcgDist()` computes, for each vertex of the first mesh, the distance to the nearest point on the surface of the second mesh:

```r
dist_result <- vcgDist(mesh_design, mesh_built_smooth)

summary(dist_result$distances)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   0.000   0.021   0.089   0.143   0.198   4.782
```

11.4 Deviation Colour Map

Map the distances to a blue–white–red ramp and render the design mesh coloured by deviation. Clip the colour scale at 0.5 m. Anything beyond that is shown as maximum red so extreme outliers don't wash out the detail:

```r
library(rgl)

cols      <- colorRampPalette(c("#2166ac", "#f7f7f7", "#d6604d"))(100)
d_norm    <- pmin(dist_result$distances / 0.5, 1)
vert_cols <- cols[ceiling(d_norm * 99) + 1]

open3d()
shade3d(mesh_design, col = vert_cols)
rglwidget()
```

11.5 Summary Statistics

```r
d <- dist_result$distances

deviation_summary <- data.frame(
  metric = c("Max deviation (m)", "Mean deviation (m)",
             "RMS deviation (m)", "% vertices > 50 mm",
             "% vertices > 100 mm"),
  value  = round(c(max(d), mean(d), sqrt(mean(d^2)),
                   100 * mean(d > 0.05),
                   100 * mean(d > 0.10)), 3)
)
deviation_summary
#>                 metric  value
#> 1    Max deviation (m)  4.782
#> 2   Mean deviation (m)  0.143
#> 3    RMS deviation (m)  0.221
#> 4  % vertices > 50 mm 34.200
#> 5 % vertices > 100 mm 18.700
```

Tip

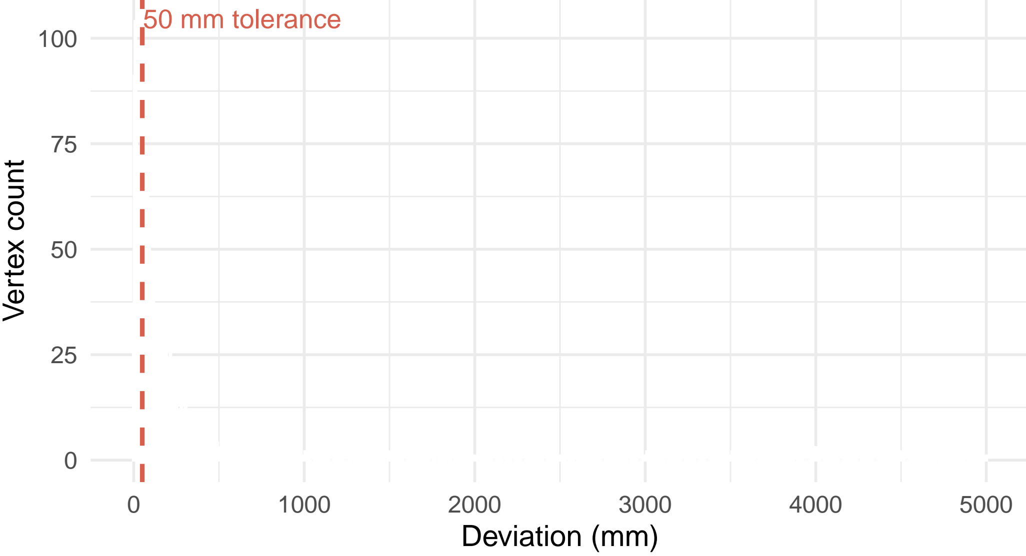
A mean deviation below 50 mm (0.05 m) is generally acceptable for architectural construction tolerances. Structural and civil works often require tighter thresholds. Always check the project specification before drawing conclusions.

11.6 Deviation Histogram

A histogram puts the distribution in context and makes it easy to spot whether deviations are concentrated in one area or spread uniformly across the model:

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

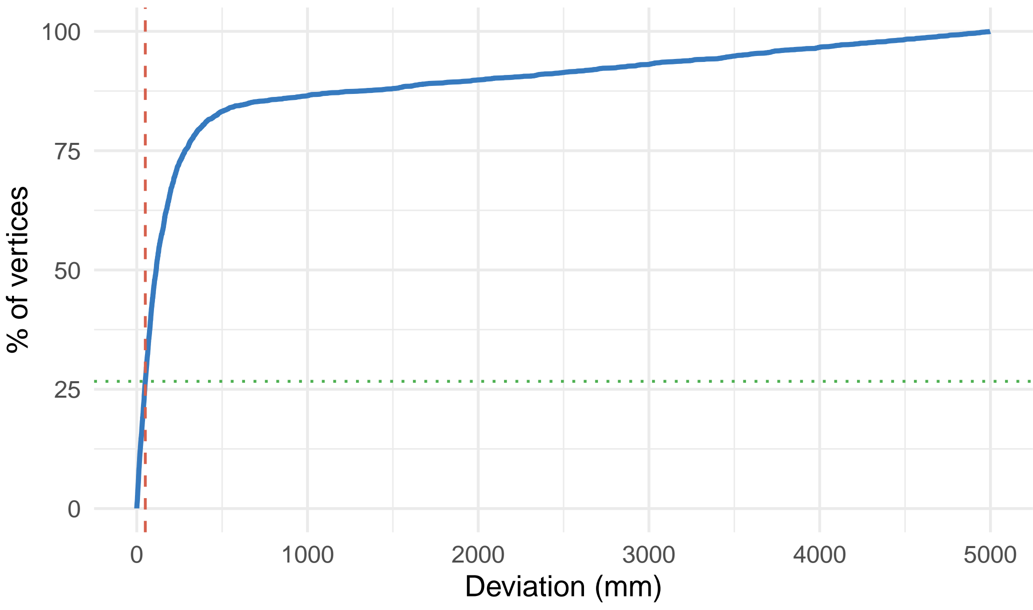
```
ggplot(data.frame(deviation_mm = d * 1000), aes(x = deviation_mm)) +
  geom_histogram(binwidth = 5, fill = "#367ABF", colour = "white") +
  geom_vline(xintercept = 50, linetype = "dashed", colour = "#d6604d", linewidth = 0.8) +
  annotate("text", x = 55, y = Inf, label = "50 mm tolerance",
           hjust = 0, vjust = 1.5, colour = "#d6604d", size = 3.5) +
  labs(title = "As-Designed vs. As-Built Deviation",
       subtitle = "Per-vertex surface distance",
       x = "Deviation (mm)",
       y = "Vertex count") +
  theme_minimal()
```



11.7 Cumulative Distribution

A CDF plot tells you what fraction of the model is within any given tolerance, handy for pass/fail reporting:

```
ggplot(data.frame(deviation_mm = sort(d * 1000)),
       aes(x = deviation_mm, y = seq_along(deviation_mm) / length(d) * 100)) +
  geom_line(colour = "#367ABF", linewidth = 0.9) +
  geom_vline(xintercept = 50, linetype = "dashed", colour = "#d6604d") +
  geom_hline(yintercept = 100 * mean(d <= 0.05),
             linetype = "dotted", colour = "#4CAF50") +
  labs(title = "Cumulative Deviation Distribution",
       x = "Deviation (mm)",
       y = "% of vertices") +
  theme_minimal()
```



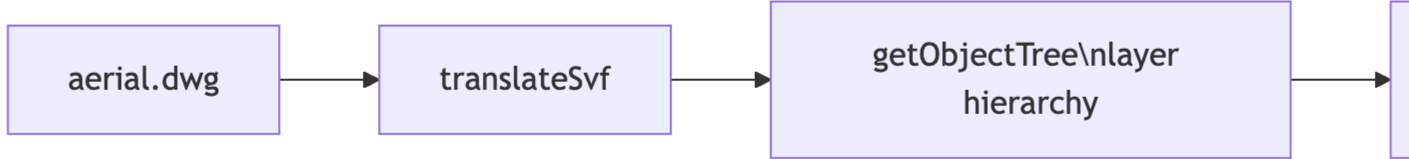
Part V

DWG & DXF Analytics

12 Layer Structure Analysis

Every DWG and DXF file organises its geometry into named layers. Unlock those layers through the Model Derivative API and you've got a structured dataset ready for `dplyr`. This chapter extracts per-layer element counts and areas and charts them — no CAD software required.

The data flows like this:



12.1 Prerequisites

```
install.packages(c("dplyr", "ggplot2"))
```

12.2 Authenticate and Translate

SVF format is what unlocks metadata access — translate first, then query:

```
library(AutoDeskR)
library(dplyr)

resp     <- getToken(id     = Sys.getenv("client_id"),
                     secret = Sys.getenv("client_secret"),
                     scope  = "data:read data:write")
myToken  <- resp$content$access_token

myEncodedUrn <- jsonlite::base64_enc(Sys.getenv("urn"))
translateSvf(urn = myEncodedUrn, token = myToken)

repeat {
  status <- checkFile(urn = myEncodedUrn, token = myToken)
  if (status$content$status == "success") break
  Sys.sleep(5)
}
```

```r
resp_meta <- getMetadata(urn = myEncodedUrn, token = myToken)
myGuid    <- resp_meta$content$data$metadata[[1]]$guid
```

12.3 The Object Tree

`getObjectTree()` returns the model hierarchy. In a DWG, the root node's children are the layers:

```r
tree_resp <- getObjectTree(guid = myGuid, urn = myEncodedUrn, token = myToken)
root      <- tree_resp$content$data$objects[[1]]

root$objects[[1]]
#> $objectid [1] 2
#> $name     [1] "Layer: A-SITE"
#> $objects       # geometry objects on this layer
```

12.4 Flatten the Tree

A small recursive helper turns the nested list into a tidy data frame — one row per leaf object:

```r
flatten_tree <- function(node, parent_layer = NA_character_) {
  is_layer   <- grepl("^Layer:", node$name)
  layer_name <- if (is_layer) sub("^Layer: ", "", node$name) else parent_layer
  if (is.null(node$objects)) {
    return(data.frame(objectid = node$objectid,
                      name     = node$name,
                      layer    = layer_name,
                      stringsAsFactors = FALSE))
  }
  do.call(rbind, lapply(node$objects, flatten_tree, parent_layer = layer_name))
}

tree_df <- flatten_tree(root)
head(tree_df)
#>   objectid              name   layer
#> 1        5  Polyline (closed)  A-SITE
#> 2        6  Polyline (closed)  A-SITE
#> 3       12              Line  A-BLDG
```

12.5 Extract Properties with getData()

`getData()` returns the full property set for every object — including bounding box coordinates:

```r
data_resp  <- getData(guid = myGuid, urn = myEncodedUrn, token = myToken)
collection <- data_resp$content$data$collection

props_df <- lapply(collection, function(obj) {
  geom <- obj$properties$Geometry
  data.frame(
    objectid = obj$objectid,
    layer    = obj$properties[["Layer and Material"]][["Layer"]],
    bb_min_x = geom[["Bounding Box Min X"]],
    bb_min_y = geom[["Bounding Box Min Y"]],
    bb_max_x = geom[["Bounding Box Max X"]],
    bb_max_y = geom[["Bounding Box Max Y"]],
    stringsAsFactors = FALSE
  )
}) |> do.call(what = rbind)

props_df <- props_df |>
  mutate(bb_area = (bb_max_x - bb_min_x) * (bb_max_y - bb_min_y))
```

> **Warning**
>
> Complex DWGs can return thousands of objects from `getData()`. For large files, pre-filter the object tree to the layers you care about before calling it to keep response sizes manageable.

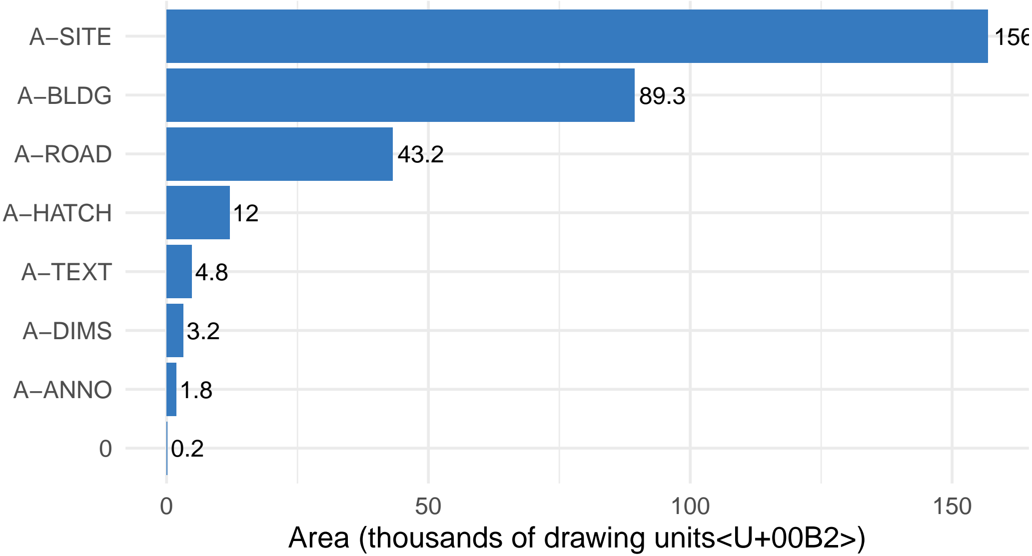
12.6 Summarise by Layer

```r
layer_summary <- props_df |>
  group_by(layer) |>
  summarise(n_objects  = n(),
            total_area = sum(bb_area, na.rm = TRUE),
            mean_area  = mean(bb_area, na.rm = TRUE),
            .groups    = "drop") |>
  arrange(desc(total_area))

layer_summary
#> # A tibble: 8 × 4
#>   layer    n_objects total_area mean_area
#>   <chr>        <int>      <dbl>     <dbl>
#> 1 A-SITE          34    156823.     4612.
#> 2 A-BLDG          21     89341.     4254.
#> 3 A-ROAD          12     43218.     3601.
#> 4 A-HATCH          8     12044.     1506.
#> 5 A-TEXT          47      4831.      103.
```

12.7 Area by Layer — Bar Chart

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```
ggplot(layer_summary,
       aes(x = reorder(layer, total_area), y = total_area / 1000)) +
  geom_col(fill = "#367ABF") +
  geom_text(aes(label = round(total_area / 1000, 1)),
            hjust = -0.1, size = 3.2) +
  coord_flip() +
  labs(title = "Total Bounding-Box Area by Layer",
       subtitle = "aerial.dwg",
       x = NULL, y = "Area (thousands of drawing units²)") +
  theme_minimal()
```

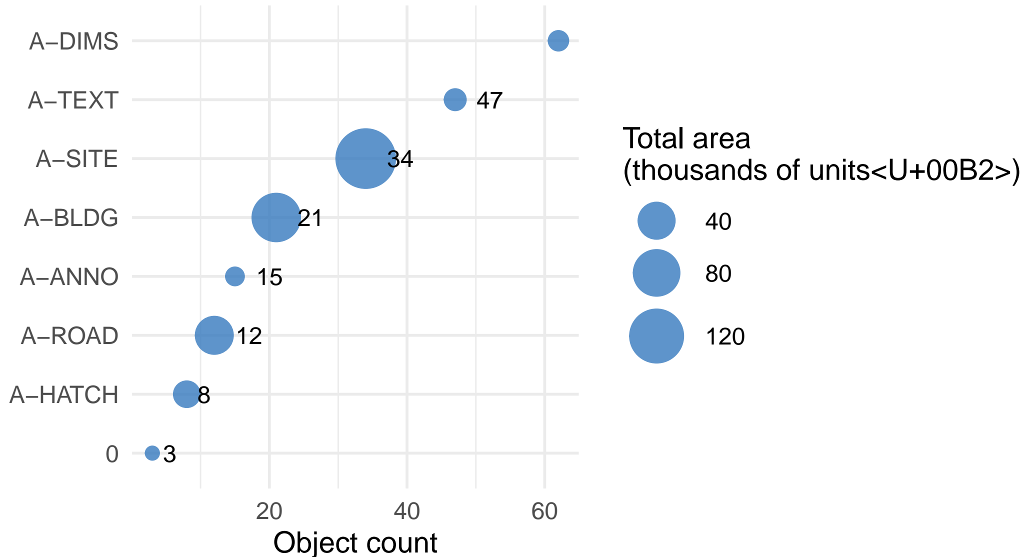


12.8 Objects per Layer — Dot Plot

A dot plot works better than a bar chart when you want to compare both count *and* area simultaneously:

```
ggplot(layer_summary, aes(x = n_objects, y = reorder(layer, n_objects))) +
  geom_point(aes(size = total_area / 1000), colour = "#367ABF", alpha = 0.8) +
  geom_text(aes(label = n_objects), hjust = -0.8, size = 3) +
  scale_size_continuous(name = "Total area\n(thousands of units²)",
                        range = c(2, 10)) +
  labs(title = "Objects per Layer",
```

```
      subtitle = "Point size proportional to total bounding-box area",
      x = "Object count", y = NULL) +
  theme_minimal()
```



The `props_df` and `layer_summary` objects carry forward into Attribute Extraction and Cross-Drawing Comparison.

13 Attribute Extraction

Layer names and bounding boxes are just the start. DWG files also carry custom properties and block attributes, zone designations, construction types, sheet numbers, column marks, that make the drawing data genuinely useful for analysis. This chapter extracts those properties from the `getData()` response and reshapes them into analysis-ready data frames.

This chapter uses `collection` from Layer Structure Analysis.

13.1 Custom Properties

Custom properties appear under non-standard category names in `obj$properties`. The standard categories to exclude are `"Layer and Material"`, `"Geometry"`, and `"General"`:

```
library(dplyr)

std_cats <- c("Layer and Material", "Geometry", "General")

custom_df <- lapply(collection, function(obj) {
  props_flat  <- unlist(obj$properties, use.names = TRUE)
  custom_keys <- names(props_flat)[
    !grepl(paste(std_cats, collapse = "|"), names(props_flat))
  ]
  if (length(custom_keys) == 0) return(NULL)
  data.frame(objectid   = obj$objectid,
             prop_name  = custom_keys,
             prop_value = props_flat[custom_keys],
             stringsAsFactors = FALSE)
}) |> do.call(what = rbind)

head(custom_df)
#>   objectid              prop_name      prop_value
#> 1       42        Custom.Zone          Zone A
#> 2       42 Custom.Construction.Type   RC Frame
#> 3       42     Custom.Floor.Area          428.5
#> 4       43        Custom.Zone          Zone B
```

13.2 Block Attributes

Block inserts carry named attributes under `obj$properties$Attributes`:

```
block_df <- lapply(collection, function(obj) {
  if (is.null(obj$properties$Attributes)) return(NULL)
  attrs <- obj$properties$Attributes
  data.frame(objectid   = obj$objectid,
             block_name = obj$name,
             attr_name  = names(attrs),
             attr_value = unlist(attrs),
             stringsAsFactors = FALSE)
}) |> do.call(what = rbind)

head(block_df)
#>   objectid  block_name   attr_name  attr_value
#> 1       88 TITLE-BLOCK   SHEET_NO      A-001
#> 2       88 TITLE-BLOCK   REVISION      Rev 3
#> 3       89  COLUMN-TAG   MARK          C1
#> 4       89  COLUMN-TAG   GRID          A/1
```

13.3 Pivot to Wide Format

Long-form property data is easy to filter but awkward to join. `pivot_wider()` gives you one column per property:

```
library(tidyr)

custom_wide <- custom_df |>
  mutate(prop_name = sub("^Custom\\.", "", prop_name)) |>
  pivot_wider(names_from  = prop_name,
              values_from = prop_value)

head(custom_wide)
#> # A tibble: 3 × 4
#>   objectid Zone   Construction.Type Floor.Area
#>      <int> <chr>  <chr>             <chr>
#> 1       42 Zone A RC Frame          428.5
#> 2       43 Zone B Steel Frame       312.0
#> 3       51 Zone A Masonry           198.3
```

13.4 Join Attributes to Layer Data

Merge custom properties back onto the geometry data frame for combined analysis:

```
enriched <- props_df |>
  left_join(custom_wide, by = "objectid")
```

13.5 Floor Area by Zone and Construction Type

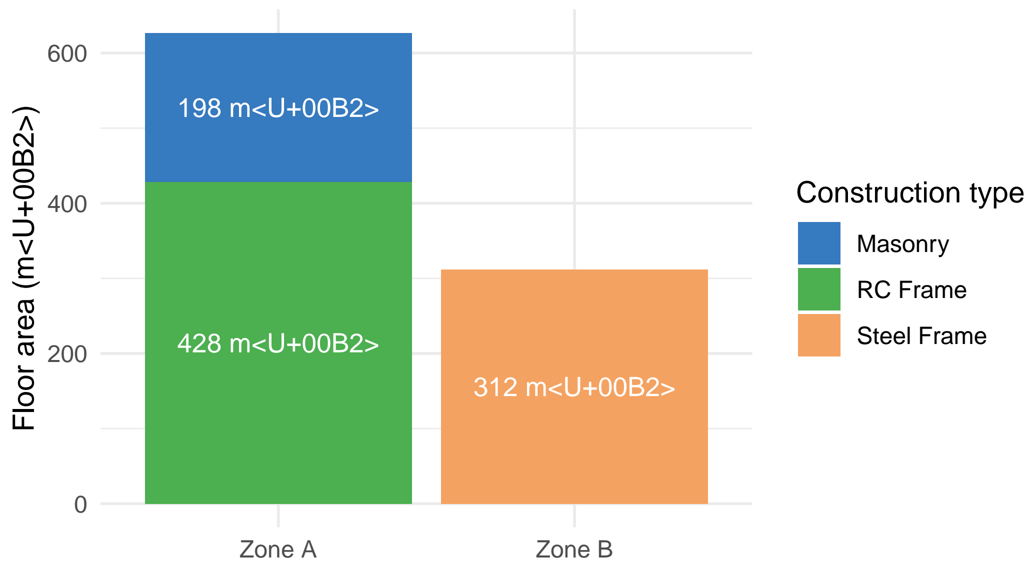
```
zone_summary <- enriched |>
  filter(layer == "A-BLDG", !is.na(Zone)) |>
  mutate(floor_area = as.numeric(Floor.Area)) |>
  group_by(Zone, Construction.Type) |>
  summarise(total_floor_area = sum(floor_area, na.rm = TRUE),
            n_units          = n(),
            .groups          = "drop")

zone_summary
#> # A tibble: 3 × 4
#>   Zone   Construction.Type total_floor_area n_units
#> 1 Zone A Masonry                      198.3       1
#> 2 Zone A RC Frame                     428.5       1
#> 3 Zone B Steel Frame                  312.0       1
```

13.6 Visualising Custom Attributes

A stacked bar chart of floor area by zone and construction type:

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```
ggplot(zone_summary,
       aes(x = Zone, y = total_floor_area, fill = Construction.Type)) +
  geom_col(position = "stack") +
  geom_text(aes(label = paste0(round(total_floor_area, 0), " m²")),
            position = position_stack(vjust = 0.5),
            colour = "white", size = 3.5) +
  scale_fill_manual(values = c("#367ABF", "#4CAF50", "#F4A261")) +
  labs(title = "Floor Area by Zone and Construction Type",
       subtitle = "Extracted from block attributes - aerial.dwg",
       x = NULL, y = "Floor area (m²)", fill = "Construction type") +
  theme_minimal()
```



13.7 Export

Write the enriched data frame to CSV for handover or further processing:

```
write.csv(enriched, file = "drawing_attributes.csv", row.names = FALSE)
```

14 Cross-Drawing Comparison

Got a drawing set? Multiple revisions? Discipline packages from different consultants? This chapter shows how to compare their layer structures systematically, what layers were added, what was removed, and where the object counts changed most between versions.

14.1 Upload Multiple Drawings

Upload each DWG to the same OSS bucket (see Data Management):

```
library(AutoDeskR)
library(dplyr)

resp    <- getToken(id     = Sys.getenv("client_id"),
                    secret = Sys.getenv("client_secret"),
                    scope  = "data:read data:write")
myToken <- resp$content$access_token

dwg_files <- c("floor_plan_v1.dwg", "floor_plan_v2.dwg", "site_plan.dwg")

urns <- sapply(dwg_files, function(f) {
  resp <- uploadFile(file   = f,
                     token  = myToken,
                     bucket = Sys.getenv("bucket"))
  resp$content$objectId
})
```

14.2 Extract Layer Sets

A helper function wraps the translate → poll → tree chain for one file and returns its layer names:

```
get_layers <- function(urn, token) {
  enc_urn <- jsonlite::base64_enc(urn)
  translateSvf(urn = enc_urn, token = token)
  repeat {
    if (checkFile(urn = enc_urn, token = token)$content$status == "success") break
    Sys.sleep(5)
  }
  meta <- getMetadata(urn = enc_urn, token = token)
```

```r
  guid <- meta$content$data$metadata[[1]]$guid
  tree <- getObjectTree(guid = guid, urn = enc_urn, token = token)
  root <- tree$content$data$objects[[1]]
  layer_nodes <- Filter(function(n) grepl("^Layer:", n$name), root$objects)
  sub("^Layer: ", "", vapply(layer_nodes, `[[`, character(1), "name"))
}

layer_sets <- lapply(urns, get_layers, token = myToken)
```

14.3 What Changed?

`setdiff()` pinpoints additions and removals:

```r
added   <- setdiff(layer_sets[["floor_plan_v2.dwg"]],
                   layer_sets[["floor_plan_v1.dwg"]])
removed <- setdiff(layer_sets[["floor_plan_v1.dwg"]],
                   layer_sets[["floor_plan_v2.dwg"]])

cat("Added:  ", paste(added,   collapse = ", "), "\n")
#> Added:   A-FURN-NEW, A-ELEC-EV
cat("Removed:", paste(removed, collapse = ", "), "\n")
#> Removed: A-TEMP-SITE
```

14.4 Presence Matrix

Build a tidy data frame showing which layers appear in which drawings:

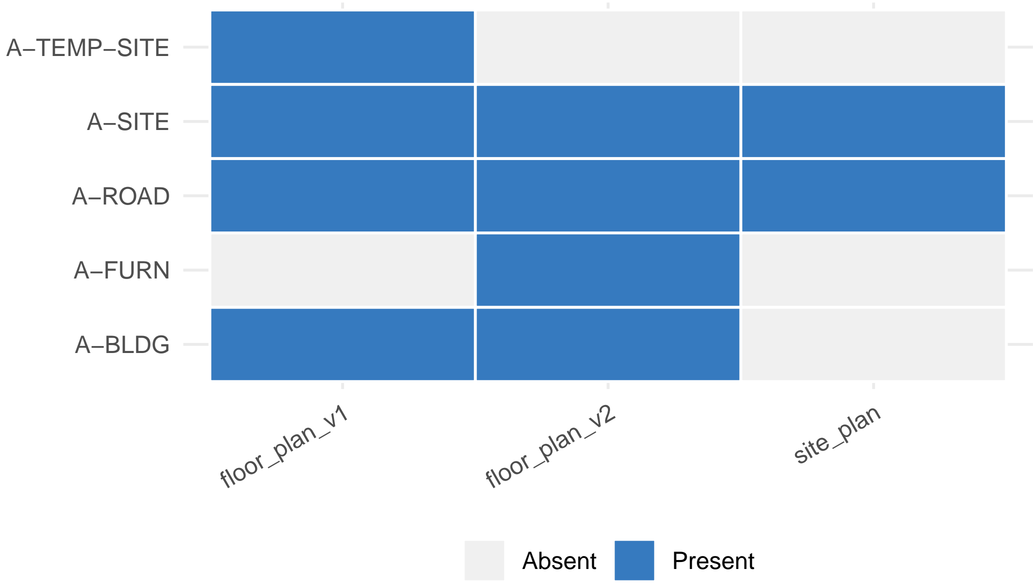
```r
all_layers   <- unique(unlist(layer_sets))
presence_df  <- data.frame(layer = all_layers)

for (nm in names(layer_sets)) {
  presence_df[[nm]] <- all_layers %in% layer_sets[[nm]]
}
```

14.5 Layer Presence Heatmap

A heatmap is the clearest way to show presence/absence across many drawings:

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```r
ggplot(heat_df, aes(x = drawing, y = layer, fill = present)) +
  geom_tile(colour = "white", linewidth = 0.5) +
  scale_fill_manual(values  = c("FALSE" = "#f0f0f0", "TRUE" = "#367ABF"),
                    labels  = c("FALSE" = "Absent", "TRUE" = "Present")) +
  labs(title = "Layer Presence Across Drawings",
       x = NULL, y = NULL, fill = NULL) +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 30, hjust = 1),
        legend.position = "bottom")
```



14.6 Object Count Changes

Go one level deeper: not just whether a layer exists, but how many objects it contains and how that changed:

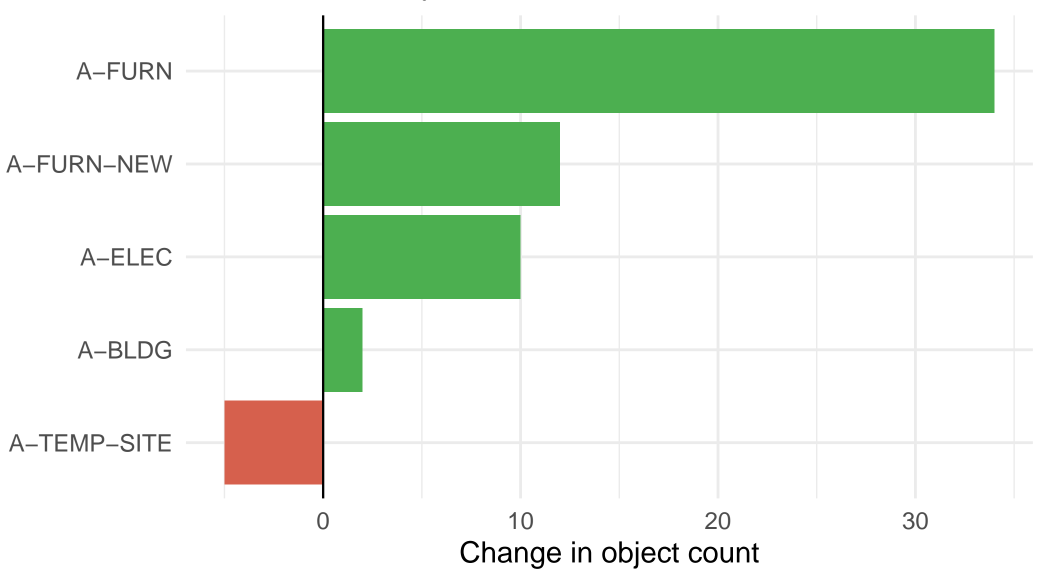
```r
get_layer_counts <- function(urn, token) {
  enc_urn   <- jsonlite::base64_enc(urn)
  meta      <- getMetadata(urn = enc_urn, token = token)
  guid      <- meta$content$data$metadata[[1]]$guid
  data_resp <- getData(guid = guid, urn = enc_urn, token = token)
  lapply(data_resp$content$data$collection, function(obj) {
    data.frame(layer = obj$properties[["Layer and Material"]][["Layer"]],
               stringsAsFactors = FALSE)
  }) |> do.call(what = rbind) |> count(layer, name = "n_objects")
}
```

```r
counts_v1 <- get_layer_counts(urns[["floor_plan_v1.dwg"]], myToken)
counts_v2 <- get_layer_counts(urns[["floor_plan_v2.dwg"]], myToken)

comparison <- full_join(counts_v1, counts_v2, by = "layer",
                        suffix = c("_v1", "_v2")) |>
  mutate(change = replace_na(n_objects_v2, 0L) -
                  replace_na(n_objects_v1, 0L)) |>
  arrange(desc(abs(change)))

comparison
#>          layer n_objects_v1 n_objects_v2 change
#> 1       A-FURN           NA           34     34
#> 2   A-FURN-NEW           NA           12     12
#> 3     A-ELEC            8           18     10
#> 4     A-BLDG           21           23      2
#> 5  A-TEMP-SITE            5           NA     -5
```

14.7 Change Waterfall Chart

A waterfall chart makes the direction and magnitude of each change immediately readable:

```r
ggplot(comparison,
       aes(x = reorder(layer, change),
           y = change,
           fill = change > 0)) +
  geom_col(show.legend = FALSE) +
  geom_hline(yintercept = 0, linewidth = 0.4) +
  scale_fill_manual(values = c("TRUE" = "#4CAF50", "FALSE" = "#d6604d")) +
  coord_flip() +
  labs(title = "Object Count Changes: v1 → v2",
       subtitle = "Green = added objects, red = removed",
       x = NULL, y = "Change in object count") +
  theme_minimal()
```


Object Count Changes: v1 <U+2192> v2
Green = added objects, red = removed
A–FURN
A–FURN–NEW
A–ELEC
A–BLDG
A–TEMP–SITE
0
10
20
30
Change in object count

15 Automated Drawing Reports

You've got the layer summary, the attribute data, the charts. Now let's wrap it all into a polished, reproducible report that regenerates automatically whenever a new DWG lands in the bucket. This chapter combines `gt` (Iannone et al. 2024) tables with Quarto's parameterised rendering to produce professional HTML reports with one function call.

15.1 A gt Summary Table

`gt` turns a data frame into a presentation-ready table in a few chained calls. The `layer_summary` from Layer Structure Analysis is a natural fit:

```
library(gt)

layer_summary |>
  gt() |>
  tab_header(
    title    = "Drawing Layer Summary",
    subtitle = "aerial.dwg - AutoDeskR extraction"
  ) |>
  cols_label(
    layer      = "Layer",
    n_objects  = "Objects",
    total_area = "Total Area",
    mean_area  = "Mean Area"
  ) |>
  fmt_number(columns = c(total_area, mean_area), decimals = 1) |>
  tab_style(
    style     = cell_fill(color = "#EBF3FB"),
    locations = cells_column_labels()
  ) |>
  tab_source_note(
    source_note = paste("Extracted", Sys.Date(), "via AutoDeskR v0.4.0")
  ) |>
  tab_options(table.font.size = "small")
```

15.2 Parameterised Reports

A parameterised Quarto document accepts different inputs at render time, making it reusable for any DWG without editing the source file. Add a `params` block to the YAML:

```
---
title: "Drawing Report"
format: html
params:
  urn: "YOUR_URN_HERE"
  drawing_name: "aerial.dwg"
---
```

Access the parameters inside the document:

```
urn          <- params$urn
drawing_name <- params$drawing_name
```

Use `drawing_name` in headings so each rendered report is self-labelled:

```
cat("## Layer Summary:", drawing_name, "\n")
```

15.3 Render Programmatically

`quarto::quarto_render()` renders the parameterised template to a named output file. The template should be a **separate `.qmd` file** (e.g., `report-template.qmd`) that contains the YAML `params` block and the analysis code — not this book chapter. Save the parameterised document as `report-template.qmd`, then loop over your drawing list:

```
library(AutoDeskR)
library(quarto)

# Build a named list of URNs from the objects in your bucket
resp  <- listObjects(token = myToken, bucket = "my-project-bucket")
items <- resp$content$items   # data frame: objectKey, objectId, size

urns <- setNames(items$objectId, items$objectKey)
# urns[["site_plan.dwg"]]      → "urn:adsk.objects:os.object:..."
# urns[["floor_plan_v2.dwg"]]  → "urn:adsk.objects:os.object:..."

drawings <- list(
  list(urn = urns[["site_plan.dwg"]],     name = "site_plan"),
  list(urn = urns[["floor_plan_v2.dwg"]], name = "floor_plan_v2")
)
```

```r
for (dwg in drawings) {
  quarto_render(
    input          = "report-template.qmd",   # the template, not this chapter
    output_file    = paste0("report_", dwg$name, ".html"),
    execute_params = list(
      urn          = dwg$urn,
      drawing_name = paste0(dwg$name, ".dwg")
    )
  )
  cat("Generated: report_", dwg$name, ".html\n", sep = "")
}
#> Generated: report_site_plan.html
#> Generated: report_floor_plan_v2.html
```

15.4 Scheduling Reports

Keep reports fresh automatically by triggering `quarto_render()` on a schedule.

GitHub Actions — run on a cron or whenever DWGs are pushed:

```yaml
# .github/workflows/drawing-report.yml
on:
  schedule:
    - cron: "0 6 * * 1"   # every Monday at 06:00 UTC
  push:
    paths: ["dwg-files/**"]

jobs:
  render:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: r-lib/actions/setup-r@v2
      - run: Rscript -e 'quarto::quarto_render("drawing-reports.qmd")'
```

cronR (local) — schedule from a local R session:

```r
library(cronR)
cmd <- cron_rscript("render_reports.R")
cron_add(command     = cmd,
         frequency   = "weekly",
         at          = "06:00",
         id          = "drawing_report",
         description = "Weekly DWG report")
```

> **Tip**
>
> Store credentials (`client_id`, `client_secret`, bucket name) as GitHub Actions secrets or in `~/.Renviron` on the local machine. Never hardcode them in the report source file. Your future self will thank you.

Part VI

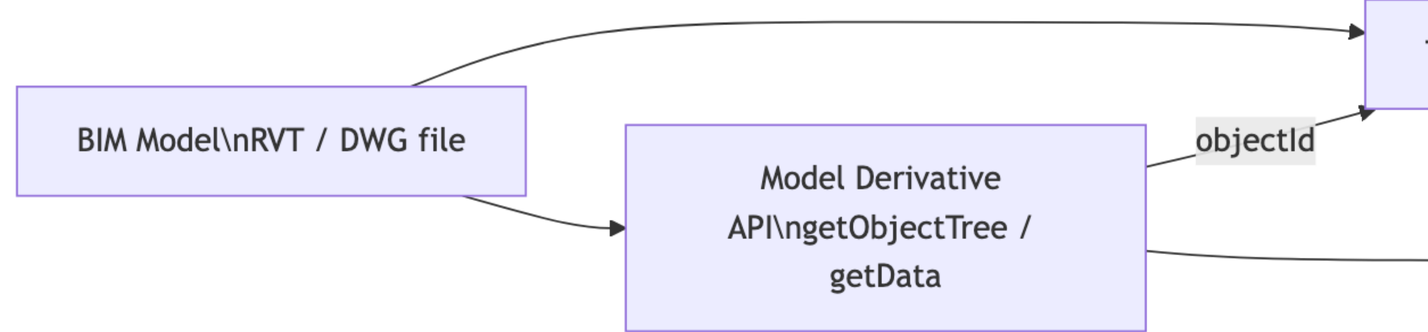
Digital Twins

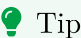
16 AutoDesk Tandem Overview

A BIM model is a snapshot of a building at design time. A **digital twin** takes that same model and connects it to the building's live operational data — temperature sensors, occupancy counters, energy meters — so you can see what's actually happening inside, not just what was drawn. AutoDesk Tandem is APS's platform for building and querying digital twins.

This chapter introduces Tandem concepts and shows how to talk to the Tandem REST API directly from R using `httr2` (Wickham 2024) and a standard APS token from `getToken()`. See (Autodesk, Inc. 2024) for the full API reference.

The relationship between a BIM model and a Tandem twin looks like this:



 Tip

Tandem requires an active **AutoDesk Tandem subscription** in addition to an APS app registration. Authentication uses the same OAuth Bearer token as the rest of this book — no extra credential setup needed.

16.1 Twin vs. BIM Model

	BIM Model	Digital Twin
Primary data	Geometry + properties	Geometry + live sensor streams
Update frequency	Per design revision	Continuous / real-time
Primary use	Design & construction	Operations & maintenance
APS API	Model Derivative	Tandem REST
R entry point	`getObjectTree()`, `getData()`	`httr2::request()`

16.2 Authentication

The Tandem API accepts the same Bearer token as all other APS services:

```
library(AutoDeskR)
library(httr2)

resp    <- getToken(id     = Sys.getenv("client_id"),
                    secret = Sys.getenv("client_secret"),
                    scope  = "data:read")
myToken <- resp$content$access_token
```

16.3 Listing Facilities

A *facility* in Tandem is the digital twin of a building or site — the top-level container for model data and sensor streams. List all facilities your token can access:

```
facilities_resp <- request("https://tandem.autodesk.com/api/v1/groups") |>
  req_auth_bearer_token(myToken) |>
  req_perform()

facilities <- resp_body_json(facilities_resp)

facilities_df <- lapply(facilities, function(f) {
  data.frame(id = f$id, name = f$name, stringsAsFactors = FALSE)
}) |> do.call(what = rbind)

facilities_df
#>                                      id              name
#> 1  urn:adsk.dtdm:facility.abc123def456  Office Block A
#> 2  urn:adsk.dtdm:facility.xyz789uvw012  Warehouse Site B
```

16.4 Facility Details

```
facilityId <- "urn:adsk.dtdm:facility.abc123def456"

facility_resp <- request(
  paste0("https://tandem.autodesk.com/api/v1/twins/", facilityId)
) |>
  req_auth_bearer_token(myToken) |>
  req_perform()

facility <- resp_body_json(facility_resp)
```

```r
cat("Name:        ", facility$displayName, "\n")
#> Name:         Office Block A
cat("Created:     ", facility$createTime, "\n")
#> Created:      2025-11-04T09:22:14Z
cat("Linked URNs:", length(facility$links), "\n")
#> Linked URNs:  2
```

`facility$links` contains the APS Model Derivative URNs of the BIM models embedded in this twin — the same URNs you'd pass to `getObjectTree()` or `getData()`, which is what makes the `objectId` the key that links the two worlds together.

16.5 Looking Up an Element

Any `objectId` from `getData()` can be looked up in Tandem to find the corresponding physical element and its classification:

```r
element_resp <- request(
  paste0("https://tandem.autodesk.com/api/v1/twins/",
         facilityId, "/elements/", 42L)   # objectId 42
) |>
  req_auth_bearer_token(myToken) |>
  req_perform()

element <- resp_body_json(element_resp)
cat("Name:          ", element$displayName, "\n")
#> Name:           Level 2 HVAC Unit
cat("Classification:", element$classification, "\n")
#> Classification: MEP:HVAC:Air Handling Unit
```

The Linking Sensor Streams chapter shows how to pull time-series readings for elements like this one.

17 Linking Sensor Streams

Each element in a Tandem facility can have sensor streams attached to it — named time-series channels that report temperature, CO , occupancy, energy consumption, or whatever the building management system sends. Because streams are keyed to the same `objectId` that `getObjectTree()` and `getData()` use, joining live sensor data to BIM metadata is just a regular table join.

This chapter uses the facility set up in AutoDesk Tandem Overview.

17.1 Listing Streams for a Facility

```r
library(AutoDeskR)
library(httr2)
library(dplyr)

resp       <- getToken(id     = Sys.getenv("client_id"),
                       secret = Sys.getenv("client_secret"),
                       scope  = "data:read")
myToken    <- resp$content$access_token
facilityId <- "urn:adsk.dtdm:facility.abc123def456"

streams_resp <- request(
  paste0("https://tandem.autodesk.com/api/v1/twins/", facilityId, "/streams")
) |>
  req_auth_bearer_token(myToken) |>
  req_perform()

streams <- resp_body_json(streams_resp)

streams_df <- lapply(streams, function(s) {
  data.frame(stream_id    = s$id,
             display_name = s$displayName,
             element_id   = s$elementId,   # objectId of the attached model element
             unit         = s$unit,
             last_reading = s$lastValue,
             stringsAsFactors = FALSE)
}) |> do.call(what = rbind)

streams_df
#>             stream_id    display_name element_id  unit last_reading
```

```
#> 1  strm:temp-hvac-l2  HVAC Temp L2          42    °C         22.4
#> 2  strm:co2-office-3  CO  Office 3         108   ppm        623.0
#> 3  strm:occ-lobby     Lobby Occupancy       15   pers        14.0
#> 4  strm:energy-main   Main Energy          201   kWh       8412.5
```

17.2 Fetching Time-Series Readings

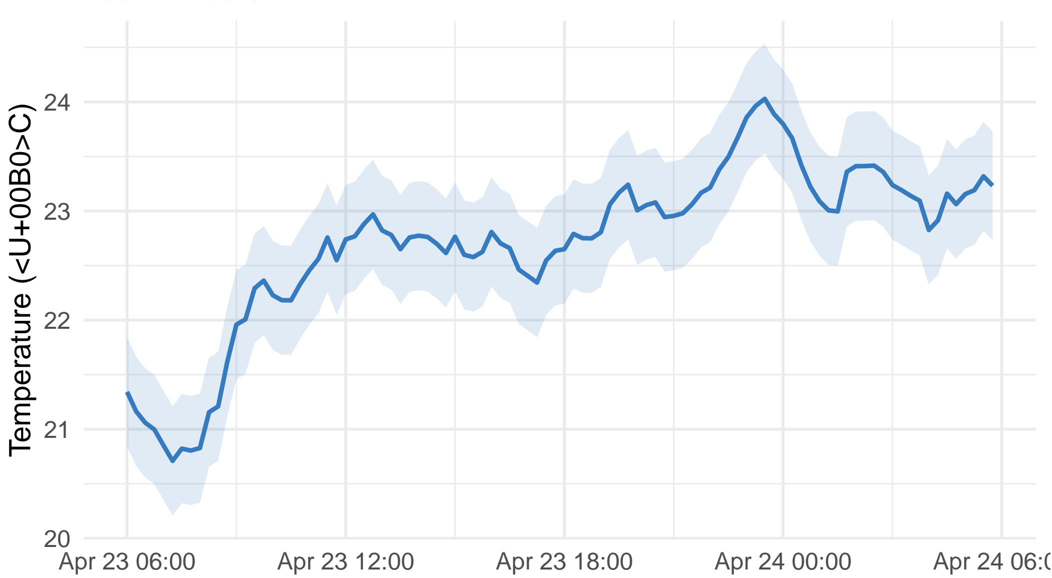
Request a time window using ISO 8601 timestamps as query parameters:

```r
streamId   <- "strm:temp-hvac-l2"
end_time   <- Sys.time()
start_time <- end_time - 86400   # last 24 hours

readings_resp <- request(
  paste0("https://tandem.autodesk.com/api/v1/twins/",
         facilityId, "/streams/", streamId)
) |>
  req_url_query(
    start = format(start_time, "%Y-%m-%dT%H:%M:%SZ", tz = "UTC"),
    end   = format(end_time,   "%Y-%m-%dT%H:%M:%SZ", tz = "UTC")
  ) |>
  req_auth_bearer_token(myToken) |>
  req_perform()

readings <- resp_body_json(readings_resp)

readings_df <- data.frame(
  timestamp = as.POSIXct(vapply(readings$values, `[[`, character(1), "t"),
                         format = "%Y-%m-%dT%H:%M:%SZ", tz = "UTC"),
  value     = vapply(readings$values, `[[`, numeric(1), "v"),
  stringsAsFactors = FALSE
)

head(readings_df)
#>             timestamp value
#> 1 2026-04-23 06:00:00  21.8
#> 2 2026-04-23 06:15:00  22.1
#> 3 2026-04-23 06:30:00  22.4
```

17.3 Plotting Readings

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```r
ggplot(readings_df, aes(x = timestamp, y = value)) +
  geom_line(colour = "#367ABF", linewidth = 0.8) +
  geom_ribbon(aes(ymin = value - 0.5, ymax = value + 0.5),
              fill = "#367ABF", alpha = 0.15) +
  labs(title    = "HVAC Temperature — Level 2",
       subtitle = "Last 24 hours",
       x = NULL, y = "Temperature (°C)") +
  theme_minimal()
```



17.4 Joining Sensor Data to Model Metadata

`element_id` in the streams response equals `objectid` from `getData()`. A left join links live readings to layer, classification, and any custom attributes:

```r
# props_df from the Layer Structure Analysis chapter

joined <- streams_df |>
  inner_join(props_df |> select(objectid, layer),
             by = c("element_id" = "objectid"))

joined
#>            stream_id    display_name element_id   unit layer
#> 1  strm:temp-hvac-l2  HVAC Temp L2           42     °C  M-HVAC
#> 3  strm:occ-lobby     Lobby Occupancy        15   pers A-LBBY
```

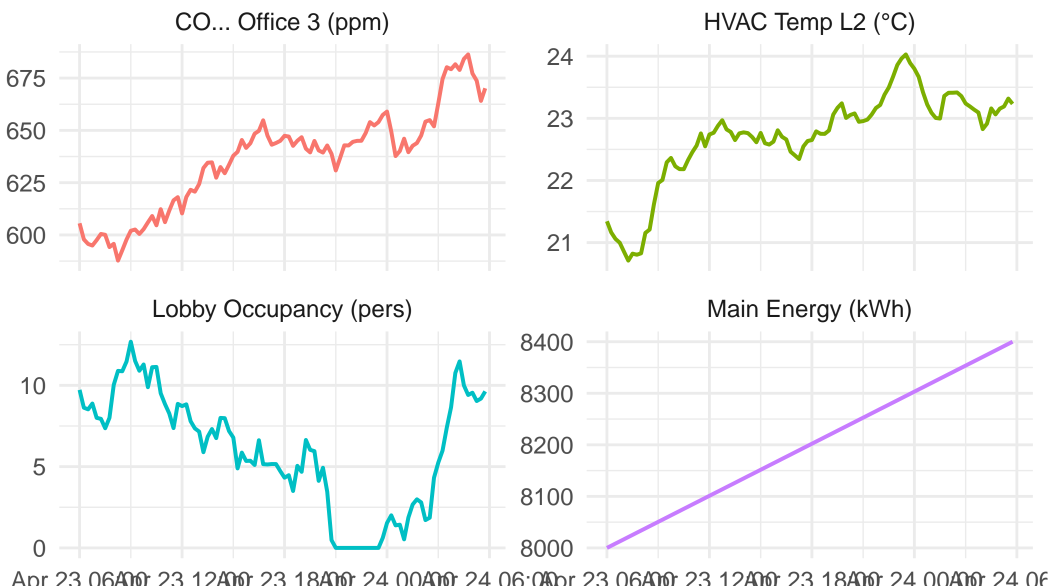
17.5 Fetching All Streams at Once

Loop over the stream list to build a single tidy data frame of all readings:

```
all_readings <- lapply(streams_df$stream_id, function(sid) {
  resp <- request(
    paste0("https://tandem.autodesk.com/api/v1/twins/",
           facilityId, "/streams/", sid)
  ) |>
    req_url_query(
      start = format(start_time, "%Y-%m-%dT%H:%M:%SZ", tz = "UTC"),
      end   = format(end_time,   "%Y-%m-%dT%H:%M:%SZ", tz = "UTC")
    ) |>
    req_auth_bearer_token(myToken) |>
    req_perform()
  readings <- resp_body_json(resp)
  data.frame(stream_id = sid,
             timestamp = as.POSIXct(
               vapply(readings$values, `[[`, character(1), "t"),
               format = "%Y-%m-%dT%H:%M:%SZ", tz = "UTC"),
             value     = vapply(readings$values, `[[`, numeric(1), "v"),
             stringsAsFactors = FALSE)
}) |> do.call(what = rbind)

# Join display names and units
all_readings <- all_readings |>
  left_join(streams_df |> select(stream_id, display_name, unit),
            by = "stream_id")
```

17.6 Multi-Stream Overview Plot

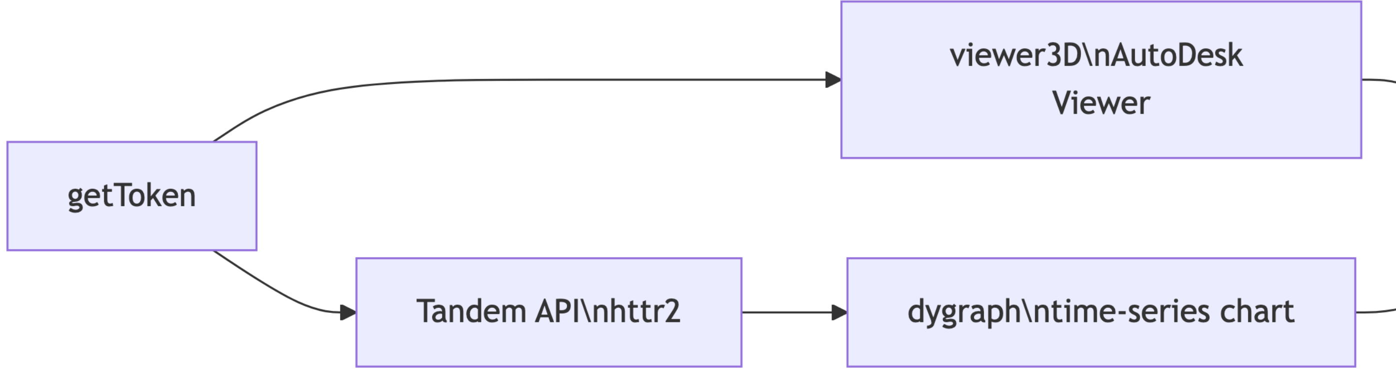
```
ggplot(all_readings, aes(x = timestamp, y = value, colour = display_name)) +
  geom_line(linewidth = 0.7) +
  facet_wrap(~ paste0(display_name, " (", unit, ")"),
             scales = "free_y", ncol = 2) +
  labs(title = "All Sensor Streams - Last 24 Hours",
       x = NULL, y = NULL) +
  theme_minimal() +
  theme(legend.position = "none")
```



The `all_readings` data frame feeds directly into the interactive Shiny dashboard in Live Dashboards.

18 Live Dashboards

Here's where everything comes together: the 3D viewer from the Viewer chapter on the left, live sensor charts from Linking Sensor Streams on the right, all in one Shiny app that refreshes automatically. This is what a digital twin dashboard looks like in R.



18.1 Required Packages

```
install.packages(c("shiny", "dygraphs", "xts", "httr2"))
```

18.2 Helper: Fetch Stream Readings

Define a reusable function that the Shiny server calls reactively:

```
library(AutoDeskR)
library(httr2)
library(dygraphs)
library(xts)
library(shiny)

get_stream_readings <- function(facility_id, stream_id, hours_back = 24) {
  token <- getToken(
    id     = Sys.getenv("client_id"),
    secret = Sys.getenv("client_secret"),
    scope  = "data:read"
  )$content$access_token
```

```
  end_time   <- Sys.time()
  start_time <- end_time - hours_back * 3600
  resp <- request(
    paste0("https://tandem.autodesk.com/api/v1/twins/",
           facility_id, "/streams/", stream_id)
  ) |>
    req_url_query(
      start = format(start_time, "%Y-%m-%dT%H:%M:%SZ", tz = "UTC"),
      end   = format(end_time,   "%Y-%m-%dT%H:%M:%SZ", tz = "UTC")
    ) |>
    req_auth_bearer_token(token) |>
    req_perform()
  readings <- resp_body_json(resp)
  data.frame(
    timestamp = as.POSIXct(vapply(readings$values, `[[`, character(1), "t"),
                           format = "%Y-%m-%dT%H:%M:%SZ", tz = "UTC"),
    value     = vapply(readings$values, `[[`, numeric(1), "v"),
    stringsAsFactors = FALSE
  )
}
```

18.3 App Startup

Fetch the token and stream list once, before the UI and server are defined:

```
myEncodedUrn <- jsonlite::base64_enc(Sys.getenv("urn"))
myToken      <- getToken(id     = Sys.getenv("client_id"),
                         secret = Sys.getenv("client_secret"),
                         scope  = "data:read")$content$access_token
facilityId   <- "urn:adsk.dtdm:facility.abc123def456"

streams_resp <- request(
  paste0("https://tandem.autodesk.com/api/v1/twins/", facilityId, "/streams")
) |>
  req_auth_bearer_token(myToken) |>
  req_perform()
streams <- resp_body_json(streams_resp)

stream_choices <- setNames(
  vapply(streams, `[[`, character(1), "id"),
  vapply(streams, `[[`, character(1), "displayName")
)
```

18.4 UI Layout

Viewer on the left, controls and chart on the right:

```
ui <- fluidPage(
  titlePanel("Digital Twin Dashboard"),
  fluidRow(
    column(7,
      viewer3D(urn        = myEncodedUrn,
               token      = myToken,
               viewerType = "headless")
    ),
    column(5,
      selectInput("stream_id", "Sensor stream:",
                  choices = stream_choices),
      sliderInput("hours_back", "Time window (hours):",
                  min = 1, max = 168, value = 24, step = 1),
      dygraphOutput("sensor_plot", height = "320px"),
      tableOutput("stats_table")
    )
  )
)
```

18.5 Server Logic

A `reactiveTimer` polls Tandem every 60 seconds so the chart stays current without the user having to refresh the page:

```
server <- function(input, output, session) {

  auto_refresh <- reactiveTimer(60000)   # 60 seconds

  readings <- reactive({
    auto_refresh()   # re-run on the timer tick
    get_stream_readings(facility_id = facilityId,
                        stream_id   = input$stream_id,
                        hours_back  = input$hours_back)
  })

  output$sensor_plot <- renderDygraph({
    df   <- readings()
    ts   <- xts(df$value, order.by = df$timestamp)
    name <- names(stream_choices)[stream_choices == input$stream_id]
    dygraph(ts, main = name) |>
      dyRangeSelector() |>
      dyOptions(fillGraph = TRUE, fillAlpha = 0.15,
```

```
                drawGrid  = TRUE, colors   = "#367ABF") |>
      dyAxis("y", label = "Value")
  })

  output$stats_table <- renderTable({
    df <- readings()
    data.frame(
      Metric = c("Readings", "Mean", "Min", "Max"),
      Value  = c(nrow(df),
                 round(mean(df$value), 2),
                 round(min(df$value),  2),
                 round(max(df$value),  2))
    )
  }, striped = TRUE, bordered = TRUE, align = "lr")
}

shinyApp(ui, server)
```

> **Warning**
>
> The Shiny server fetches Tandem data from R, not from the user's browser, so CORS restrictions don't apply. The server process does need outbound HTTPS access to `tandem.autodesk.com`, so check this if you're deploying on a restricted corporate network.

18.6 Deploying to shinyapps.io

Keep credentials out of the app source. Pass them as server environment variables at deploy time:

```
library(rsconnect)

deployApp(
  appDir     = ".",
  appName    = "twin-dashboard",
  appEnvVars = list(
    client_id     = Sys.getenv("client_id"),
    client_secret = Sys.getenv("client_secret"),
    urn           = Sys.getenv("urn")
  )
)
```

Part VII

AI Integration

19 MCP Server

AI assistants are increasingly useful not just for answering questions but for *doing things*, calling APIs, reading files, running analyses. The Model Context Protocol (MCP) is the open standard that makes that possible. It defines how an AI assistant discovers and calls external tools during a conversation, in the same way a web browser discovers and calls REST APIs.

AutoDeskR functions are exactly the kind of tools an AI agent would want to call. "Summarise the layers in this DWG." "Translate this file and give me the bounding box." "Fetch the last 24 hours of HVAC sensor readings." Each of those is a handful of AutoDeskR function calls, and with an MCP server in front of them, an AI assistant can make those calls itself, without you writing a single line of R.

This chapter shows how to expose AutoDeskR as an MCP server so that Claude Desktop, VS Code Copilot, or any other MCP-compatible client can query BIM models directly.

19.1 Proposed Tools

The table below maps ten useful MCP tools to their underlying AutoDeskR functions. Each tool is a thin wrapper. It handles authentication, calls the relevant function(s), and returns a structured JSON result.

MCP Tool	Wraps	Description
`get_token`	`getToken()`	Authenticate and return a bearer token
`upload_file`	`makeBucket()` + `uploadFile()`	Create a bucket and upload a local file
`translate_file`	`translateObj()` / `translateSvf()` + `checkFile()`	Translate a file and poll until complete
`get_layer_summary`	`translateSvf()` + `getData()`	Layer-level object count and area for a DWG
`get_object_tree`	`getObjectTree()`	Model hierarchy as a JSON tree
`get_metadata`	`getMetadata()`	List available metadata GUIDs for a translated model
`download_mesh`	`getOutputUrn()` + `downloadFile()`	Download the translated OBJ/STL to a local path
`make_pdf`	`makePdf()` + `checkPdf()`	Convert a DWG to PDF via Design Automation

MCP Tool	Wraps	Description
`get_sensor_streams`	Tandem REST API	List sensor streams for a Tandem facility
`get_stream_readings`	Tandem REST API	Fetch time-series readings for a stream

19.2 Setting Up a Local MCP Server

The `mcpr` package provides the MCP transport layer for R. Install it from GitHub:

```
devtools::install_github("devOpifex/mcpr")
```

A minimal server registers tools and starts listening. Here's a working example for the `get_layer_summary` tool:

```
library(mcpr)
library(AutoDeskR)

# Authenticate once at startup using environment variables
.token <- local({
  resp <- getToken(
    id     = Sys.getenv("client_id"),
    secret = Sys.getenv("client_secret"),
    scope  = "data:read data:write bucket:create"
  )
  resp$content$access_token
})

# Register the tool
mcp_tool(
  name        = "get_layer_summary",
  description = "Return object count and total area by layer for a translated DWG URN.",
  params = list(
    urn = mcp_param("string", "The raw objectId URN returned by uploadFile()")
  ),
  handler = function(urn) {
    encoded  <- jsonlite::base64_enc(urn)
    resp     <- getData(urn = encoded, token = .token)
    objects  <- resp$content$data$objects

    layer_counts <- table(sapply(objects, `[[`, "name"))
    data.frame(layer = names(layer_counts),
               n_objects = as.integer(layer_counts))
  }
)
```

```
# Start the server (stdio transport for Claude Desktop)
mcp_serve()
```

```
#> MCP server listening on stdio
#> Registered tools: get_layer_summary
```

Add more tools by repeating the `mcp_tool()` pattern before calling `mcp_serve()`.

19.3 Connecting to Claude Desktop

Claude Desktop reads its MCP server list from a JSON config file. On macOS, open or create `~/Library/Application Support/Claude/claude_desktop_config.json` and add an entry for the AutoDeskR server:

```
{
  "mcpServers": {
    "autodeskr": {
      "command": "Rscript",
      "args": ["/path/to/autodeskr-mcp-server.R"],
      "env": {
        "client_id": "your_aps_client_id",
        "client_secret": "your_aps_client_secret"
      }
    }
  }
}
```

Restart Claude Desktop. The AutoDeskR tools will appear in the tool palette and Claude will call them automatically when relevant.

19.4 Example Agent Conversations

Once connected, you can have conversations like:

> **You:** I've uploaded `aerial.dwg` to bucket `acmecorp-dev`. The objectId is `urn:adsk.objects:os.object:acmecorp-dev/aerial.dwg`. Summarise the layers.
>
> **Claude:** *(calls `get_layer_summary`)* The drawing has 8 layers. The largest by object count is `A-DIMS` with 62 objects, followed by `A-TEXT` with 47. The largest by area is `A-SITE` at 156,823 square units.
>
> **You:** Compare the layer sets in `floor_plan_v1.dwg` and `floor_plan_v2.dwg`.

Claude: *(calls `get_layer_summary` twice, compares results)* Version 2 adds `A-FURN` (34 objects) and `A-FURN-NEW` (12 objects). `A-TEMP-SITE` was present in v1 but has been removed.

You: Fetch the last 24 hours of temperature readings from the HVAC sensor on level 2.

Claude: *(calls `get_stream_readings`)* The stream `strm:temp-hvac-l2` shows a mean of 22.1 °C over the last 24 hours, ranging from 21.4 °C to 23.0 °C. Peak temperature was at 14:30 UTC.

19.5 Security Considerations

A few things to keep in mind before deploying:

- **Credentials:** Pass `client_id` and `client_secret` via environment variables (as shown in the config above), never hardcoded in the script. The server reads them at startup via `Sys.getenv()`.
- **Token expiry:** APS tokens expire after 3600 seconds. For long-running servers, refresh the token periodically. Add a `Sys.time()` check in the handler and re-call `getToken()` if more than 50 minutes have elapsed.
- **Rate limits:** APS enforces per-app rate limits. If the agent calls tools rapidly (e.g. in a loop over many files), add `Sys.sleep(0.5)` between calls. HTTP 429 responses should trigger a backoff, not a crash.
- **Scope:** Request only the scopes the server actually needs. A read-only analysis server only needs `data:read`; it doesn't need `data:write` or `bucket:create`.

Part VIII

Reference

20 Case Study: DWG to Shiny Dashboard

This chapter ties every API together in a single narrative. Starting from a raw DWG file on disk, we will authenticate, upload, translate, extract metadata, visualise layer data with ggplot2, and embed the live 3D model in a Shiny dashboard — all from R.

The DWG file used throughout is `aerial.dwg`, the sample file bundled with AutoDeskR:

```
dwg_path <- system.file("samples/aerial.dwg", package = "AutoDeskR")
```

20.1 Step 1: Authenticate with Combined Scopes

Every API used in this case study requires a different scope. Request them all upfront in a single token:

```
library(AutoDeskR)
library(ggplot2)
library(shiny)

resp <- getToken(
  id     = Sys.getenv("client_id"),
  secret = Sys.getenv("client_secret"),
  scope  = "data:read data:write bucket:create bucket:read code:all"
)
myToken <- resp$content$access_token
#> [1] "eyJhbGciOiJSUzI1NiIsImtpZCI6IlU3c1BKNVVpOWNaVjlIR2FhX1pIeDhEd3VYSHcifQ..."
```

20.2 Step 2: Create a Bucket and Upload the DWG

```
# Create a persistent bucket
bucket <- makeBucket(token = myToken, bucket = "case-study-bucket", policy = "persistent")
bucket$content$bucketKey
#> [1] "case-study-bucket"
```

```r
# Upload the DWG
upload <- uploadFile(
  file   = dwg_path,
  token  = myToken,
  bucket = "case-study-bucket"
)
myUrn <- upload$content$objectId
upload$content$objectKey
#> [1] "aerial.dwg"
upload$content$size
#> [1] 3145728
```

20.3 Step 3: Translate to SVF

SVF is the format required by the Viewer and for metadata extraction. Encode the URN, submit the job, and wait for it to finish:

```r
myEncodedUrn <- jsonlite::base64_enc(myUrn)

# Submit translation
translateSvf(urn = myEncodedUrn, token = myToken)$content$result
#> [1] "created"

# Poll until done
repeat {
  status <- checkFile(urn = myEncodedUrn, token = myToken)
  cat("Status:", status$content$status, "\n")
  if (status$content$status == "success") break
  Sys.sleep(10)
}
#> Status: pending
#> Status: inprogress
#> Status: inprogress
#> Status: success
```

20.4 Step 4: Extract Layer Metadata

Retrieve the GUID, then pull the full object property set to get per-layer geometry data:

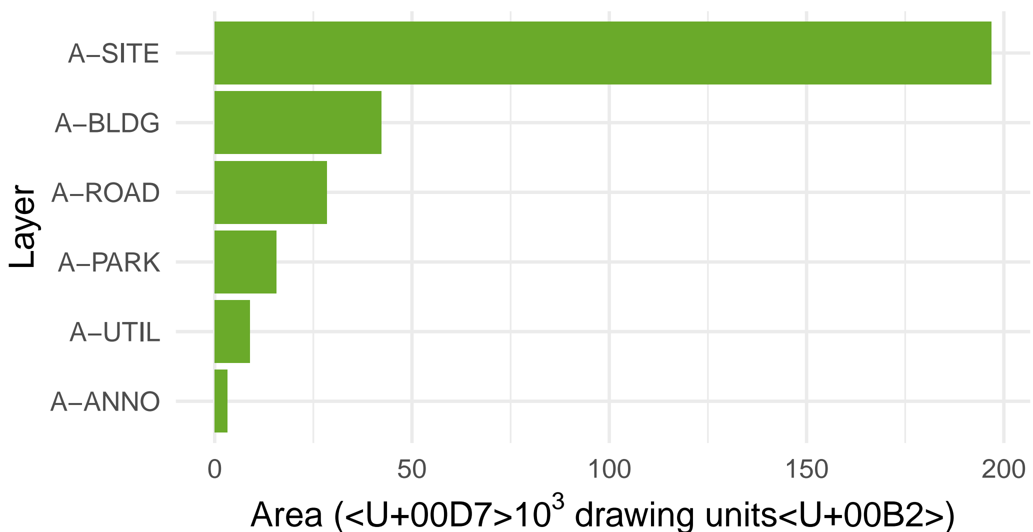
```r
# Get the viewable GUID
meta   <- getMetadata(urn = myEncodedUrn, token = myToken)
myGuid <- meta$content$data$metadata[[1]]$guid
myGuid
#> [1] "a7b3c9d2-4e1f-4a8b-b6c2-9d3e7f5a1b4c"

# Retrieve all object properties
props <- getData(guid = myGuid, urn = myEncodedUrn, token = myToken)

# Extract layer name and bounding-box area for each object
layer_data <- lapply(props$content$data$collection, function(obj) {
  layer <- obj$properties[["Layer and Material"]][["Layer"]]
  xmin  <- obj$properties$Geometry[["Bounding Box Min X"]]
  xmax  <- obj$properties$Geometry[["Bounding Box Max X"]]
  ymin  <- obj$properties$Geometry[["Bounding Box Min Y"]]
  ymax  <- obj$properties$Geometry[["Bounding Box Max Y"]]
  if (!is.null(layer) && !is.null(xmin)) {
    data.frame(
      layer = layer,
      area  = (xmax - xmin) * (ymax - ymin)
    )
  }
})

layers_df <- do.call(rbind, Filter(Negate(is.null), layer_data))
head(layers_df)
#>     layer        area
#> 1  A-SITE  196842.73
#> 2  A-BLDG   42301.18
#> 3  A-ROAD   28473.09
#> 4  A-PARK   15621.44
#> 5  A-UTIL    8904.72
#> 6  A-ANNO    3215.01
```

20.5 Step 5: Visualise with ggplot2

```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```r
ggplot(layer_totals, aes(x = reorder(layer, area), y = area / 1000)) +
  geom_col(fill = "#6aaa2a") +
  coord_flip() +
  labs(
    title    = "Bounding-Box Area by DWG Layer",
```

```r
    subtitle = "aerial.dwg — AutoDeskR sample file",
    x        = "Layer",
    y        = expression("Area (×10"^3 * " drawing units²)")
  ) +
  theme_minimal(base_size = 13)
```



20.6 Step 6: Build the Shiny Dashboard

Combine the ggplot2 chart and the embedded 3D viewer in a two-tab Shiny app:

```r
ui <- shiny::fluidPage(
  shiny::titlePanel("Aerial Site — AutoDeskR Case Study"),
  shiny::tabsetPanel(
    shiny::tabPanel(
      "3D Model",
      viewerUI("model", urn = myEncodedUrn, token = myToken)
    ),
    shiny::tabPanel(
      "Layer Analysis",
      shiny::plotOutput("layer_chart", height = "500px")
    )
  )
)
```

```r
server <- function(input, output, session) {
  output$layer_chart <- shiny::renderPlot({
    ggplot(layer_totals, aes(x = reorder(layer, area), y = area / 1000)) +
      geom_col(fill = "#6aaa2a") +
      coord_flip() +
      labs(
        title = "Bounding-Box Area by DWG Layer",
        x = "Layer",
        y = expression("Area (×10"^3 * " units²)")
      ) +
      theme_minimal(base_size = 13)
  })
}

shiny::shinyApp(ui, server)
```

The finished dashboard shows the AutoDesk WebGL viewer on the left tab and the ggplot2 analysis on the right, both driven by data extracted live from the same DWG file through the APS APIs.

20.7 Summary

Step	Function(s)	API
Authenticate	`getToken()`	Authentication
Create bucket + upload	`makeBucket()`, `uploadFile()`	Data Management
Translate to SVF	`translateSvf()`, `checkFile()`	Model Derivative
Extract properties	`getMetadata()`, `getData()`	Model Derivative
Visualise	`ggplot2`	—
Embed viewer	`viewerUI()`	Viewer

The full source for this case study is available at github.com/paulgovan/AutoDeskR in the `inst/examples/` directory.

21 Function Reference

Quick-lookup table for every exported AutoDeskR function. Click the **Chapter** link to jump to full documentation and worked examples.

21.1 Authentication

Function	Required Scopes	Key Parameters	Returns
getToken(id, secret, scope)	—	id, secret, scope (space-separated string)	$content$access_token, $content$expires_in

---

21.2 Data Management

Function	Required Scopes	Key Parameters	Returns
makeBucket(token, bucket, policy)	bucket:create	bucket (globally unique key), policy ("transient" / "temporary" / "persistent")	$content$bucketKey, $content$policyKey
checkBucket(token, bucket)	bucket:read	bucket	$content$bucketKey, $content$policyKey
deleteBucket(token, bucket)	bucket:delete	bucket (must be empty)	$status_code 200
uploadFile(file, token, bucket)	data:write	file (local path, 100 MB), bucket	$content$objectId (URN), $content$size
uploadFileSigned(file, token, bucket)	data:write	file (local path, > 100 MB), bucket	$content$objectId (URN)
listBuckets(token, limit, startAt, region)	bucket:read	limit (default 10), startAt (pagination key), region ("US" / "EMEA")	$content$items data frame

Function	Required Scopes	Key Parameters	Returns
`listObjects(token, bucket, limit)`	`data:read`	`bucket`, `limit`	`$content$items` data frame
`deleteObject(token, bucket, object)`	`data:write`	`bucket`, `object` (filename)	`$status_code` 200
`downloadFile(urn, output_urn, token, destfile)`	`data:read`	`urn` (encoded source), `output_urn` (encoded output), `destfile`	`$status_code` 200

21.3 Model Derivative

Function	Required Scopes	Key Parameters	Returns
`translateObj(urn, token)`	`data:read` `data:write`	`urn` (Base-64 encoded)	`$content$result` (`"created"`), `$content$urn`
`translateStl(urn, token)`	`data:read` `data:write`	`urn` (Base-64 encoded)	`$content$result`, `$content$urn`
`translateSvf(urn, token)`	`data:read` `data:write`	`urn` (Base-64 encoded)	`$content$result`, `$content$urn`
`checkFile(urn, token)`	`data:read`	`urn` (Base-64 encoded)	`$content$status` (`"pending"` / `"inprogress"` / `"success"`), `$content$progress`
`getOutputUrn(urn, token)`	`data:read`	`urn` (raw, not encoded)	`$content$derivatives[[1]]$childr`
`getMetadata(urn, token)`	`data:read`	`urn` (Base-64 encoded)	`$content$data$metadata[[1]]$guic` `$content$data$metadata[[1]]$name`
`getObjectTree(guid, urn, token)`	`data:read`	`guid`, `urn` (Base-64 encoded)	`$content$data$objects` (nested list)
`getData(guid, urn, token)`	`data:read`	`guid`, `urn` (Base-64 encoded)	`$content$data$collection` (list of objects with properties)

21.4 Design Automation

Function	Required Scopes	Key Parameters	Returns
`makePdf(source, destination, token)`	`code:all`	`source` (public URL to DWG), `destination` (public URL for output)	`$content$id` (WorkItem ID), `$content$status`
`checkPdf(id, token)`	`code:all`	`id` (WorkItem ID from `makePdf()`)	`$content$status` (`"pending"` / `"inprogress"` / `"success"`), `$content$stats`

21.5 Reality Capture

Function	Required Scopes	Key Parameters	Returns
`createPhotoscene(name, format, token)`	`data:read data:write`	`name` (scene label), `format` (`"rcm"` / `"rcs"` / `"obj"` / `"ortho"` / `"report"`)	`$content$photoscene$photosceneid`
`uploadImages(photoscene_id, files, token)`	`data:read data:write`	`photoscene_id`, `files` (character vector of local JPEG/PNG paths)	`$content$Files$file` (list of uploaded file records)
`processPhotoscene(photoscene_id, token)`	`data:read data:write`	`photoscene_id`	`$content$photoscene$progress` (`"0"` on submission)
`checkPhotoscene(photoscene_id, token)`	`data:read data:write`	`photoscene_id`	`$content$photoscene$progress` (string `"0"`–`"100"`), `$content$photoscene$progressmsg`
`waitForPhotoscene(photoscene_id, token, interval, timeout, verbose)`	`data:read data:write`	`interval` (seconds between polls, default 30), `timeout` (max wait in seconds, default 1800)	Final `checkPhotoscene` response when `progress == "100"`

21.6 Viewer

Function	Required Scopes	Key Parameters	Returns
`viewer3D(urn, token, viewerType)`	`data:read`	`urn` (Base-64 encoded), `viewerType` (`"header"` / `"headless"` / `"vr"`)	Opens viewer in browser / Shiny
`viewerUI(id, urn, token, viewerType)`	`data:read`	`id` (Shiny module ID), `urn`, `token`, `viewerType`	Shiny UI element for use in `fluidPage()`

21.7 Scope Quick Reference

Scope	Used by
`data:read`	Model Derivative, Viewer, Reality Capture (read)
`data:write`	Data Management (upload/delete), Model Derivative (translate), Reality Capture
`bucket:create`	Data Management (`makeBucket`)
`bucket:read`	Data Management (`checkBucket`, `listBuckets`)
`bucket:delete`	Data Management (`deleteBucket`)
`code:all`	Design Automation

21.8 Common Patterns

21.8.1 Base-64 encode a URN

The Model Derivative and Viewer APIs require the URN to be Base-64 encoded. Use `jsonlite::base64_enc()`:

```
myEncodedUrn <- jsonlite::base64_enc(myUrn)
```

21.8.2 Poll until a job finishes

All async jobs (`translateObj`, `translateSvf`, `makePdf`, `processPhotoscene`) follow the same pattern:

```r
repeat {
  status <- checkFile(urn = myEncodedUrn, token = myToken)
  if (status$content$status == "success") break
  Sys.sleep(10)
}
```

Use `waitForPhotoscene()` as a ready-made equivalent for Reality Capture jobs.

---

22 Supporting Packages

The chapters outside the Core APIs part rely on the following R packages. This section lists the key functions used in the book and the chapter where each appears.

22.1 httr2

Used for direct Tandem REST API calls in the Digital Twins chapters, and as the underlying HTTP engine for AutoDeskR.

Function	Description	Chapter
`request(url)`	Create a new HTTP request	Tandem Overview, Sensor Streams
`req_auth_bearer_token(req, token)`	Add a Bearer token header	Tandem Overview
`req_url_query(req, ...)`	Append query parameters	Sensor Streams
`req_perform(req)`	Execute the request	Tandem Overview
`req_verbose()`	Enable request/response tracing	Troubleshooting
`resp_body_json(resp)`	Parse response body as JSON	Sensor Streams

22.2 jsonlite

Function	Description	Chapter
`base64_enc(x)`	Base-64 encode a string (required for APS URNs)	Model Derivative, Case Study
`fromJSON(txt)`	Parse a JSON string into an R list	General

22.3 Rvcg

Mesh processing — surface area, volume, smoothing, and surface distance. (Schlager 2024)

Function	Description	Chapter
`vcgImport(file)`	Import OBJ, STL, or PLY mesh as `tmesh3d`	Reading Meshes
`vcgArea(mesh)`	Compute total surface area	Mesh Metrics
`vcgVolume(mesh)`	Compute volume (watertight meshes only)	Mesh Metrics
`vcgSmooth(mesh, iteration, lambda)`	Laplacian smoothing	Mesh Comparison
`vcgDist(mesh1, mesh2)`	Per-vertex surface distance	Mesh Comparison

22.4 rgl

Interactive 3D visualisation. (Adler, Murdoch, et al. 2024)

Function	Description	Chapter
`open3d()`	Open a new 3D graphics device	Mesh Visualisation
`shade3d(mesh, col)`	Render a mesh with per-vertex colours	Mesh Visualisation, Mesh Comparison
`rglwidget()`	Embed interactive 3D widget in HTML	Mesh Visualisation
`readOBJ(file)`	Alternative OBJ reader	Troubleshooting
`options(rgl.useNULL = TRUE)`	Suppress OpenGL window (headless/server use)	Troubleshooting

---

22.5 geometry

Computational geometry for open (non-watertight) meshes. (Roussel, Sterratt, et al. 2024)

Function	Description	Chapter
`convhulln(pts, options)`	Convex hull with area and volume (`options = "FA"`)	Mesh Metrics

22.6 rayshader

High-quality 3D rendered images. (Morgan-Wall 2024)

Function	Description	Chapter
`render_snapshot(filename)`	Save a ray-traced render to file	Mesh Visualisation

22.7 lidR

Point cloud analysis from LAS/LAZ files. (Roussel et al. 2020)

Function	Description	Chapter
`readLAS(file)`	Read a LAS or LAZ point cloud	Point Cloud Analysis
`normalize_height(pc, algorithm)`	Normalise Z to ground level	Point Cloud Analysis
`rasterize_canopy(pc, res, algorithm)`	Maximum-Z raster (canopy/roof height model)	Point Cloud Analysis
`grid_density(pc, res)`	Point density raster	Point Cloud Analysis
`voxelize_points(pc, res)`	Voxelise point cloud for volume estimation	Point Cloud Analysis
`npoints(pc)`	Total point count	Point Cloud Analysis
`area(pc)`	Survey footprint area	Point Cloud Analysis

22.8 dplyr

Data manipulation — filtering, grouping, joining. Used throughout the DWG Analytics and Digital Twins chapters.

Function	Description	Chapter
`group_by(...)`	Group rows for summarisation	Layer Analysis
`summarise(...)`	Aggregate within groups	Layer Analysis
`filter(...)`	Keep rows matching a condition	Attribute Extraction
`select(...)`	Choose columns	Sensor Streams
`mutate(...)`	Add or transform columns	Attribute Extraction

Function	Description	Chapter
`inner_join(x, y, by)`	Join keeping rows in both tables	Sensor Streams
`left_join(x, y, by)`	Join keeping all rows from left table	Sensor Streams

22.9 tidyr

Function	Description	Chapter
`pivot_wider(data, names_from, values_from)`	Reshape long data to wide format	Attribute Extraction

22.10 ggplot2

Data visualisation. Used in DWG Analytics, Digital Twins, and the Case Study.

Function	Description	Chapter
`ggplot(data, aes(...))`	Initialise a plot	Multiple
`geom_col()`	Bar chart (pre-counted values)	Layer Analysis
`geom_line()`	Line chart	Sensor Streams
`geom_histogram()`	Histogram	Point Cloud Analysis, Mesh Comparison
`geom_ribbon()`	Shaded confidence band	Sensor Streams
`geom_tile()`	Heatmap tiles	Cross-Drawing Comparison
`facet_wrap(~ var)`	Small multiples	Sensor Streams
`coord_flip()`	Swap x and y axes	Layer Analysis
`theme_minimal()`	Clean theme	Multiple

22.11 gt

Publication-quality tables for automated reports.(Iannone et al. 2024)

Function	Description	Chapter
`gt(data)`	Create a gt table from a data frame	Drawing Reports
`tab_header(title, subtitle)`	Add a title and subtitle	Drawing Reports
`fmt_number(columns, decimals)`	Format numeric columns	Drawing Reports
`cols_label(...)`	Rename column headers	Drawing Reports
`tab_source_note(note)`	Add a footnote	Drawing Reports

22.12 shiny

Interactive web applications and dashboards.

Function	Description	Chapter
`fluidPage(...)`	Standard responsive page layout	Viewer, Live Dashboards
`tabsetPanel(...)` / `tabPanel(...)`	Tabbed layout	Case Study
`renderPlot({})`	Reactive ggplot2 output	Case Study
`plotOutput(id)`	Placeholder for a plot	Case Study
`reactiveTimer(ms)`	Trigger reactive updates on a schedule	Live Dashboards
`shinyApp(ui, server)`	Launch the app	Case Study
`deployApp()`	Deploy to shinyapps.io	Live Dashboards

22.13 dygraphs

Interactive time-series plots for Shiny dashboards. (Vanderkam et al. 2018)

Function	Description	Chapter
`dygraph(data, main)`	Create an interactive time-series chart	Live Dashboards
`dyOptions(...)`	Customise colours, fill, axes	Live Dashboards
`dygraphOutput(id)`	Shiny UI placeholder	Live Dashboards
`renderDygraph({})`	Shiny server renderer	Live Dashboards

22.14 xts

Time-series objects required by `dygraphs`.

Function	Description	Chapter
`xts(data, order.by)`	Create an extensible time-series object	Live Dashboards

22.15 quarto

Programmatic report rendering.

Function	Description	Chapter
`quarto_render(input, execute_params)`	Render a `.qmd` file with injected parameters	Drawing Reports

23 Troubleshooting

Something's not working. Don't worry. APS errors are usually pretty informative once you know how to read them. This chapter is a reference for the most common problems, organised by HTTP status code and then by API area.

23.1 Error Code Quick Reference

The first thing to check is always `resp$status_code`. Here's what each code means and where to start looking:

Code	Meaning	Most likely cause	First step
400	Bad Request	Malformed request body; non-public URL in Design Automation	Check source/destination URLs; validate parameters
401	Unauthorized	Token expired or missing; wrong scope	Re-call `getToken()` with correct scopes
403	Forbidden	Free-tier quota exceeded; API not enabled for app	Check quota in APS portal; enable the API on your app
404	Not Found	Wrong URN; object or bucket deleted; bad GUID	Re-check URN encoding; confirm the resource exists
409	Conflict	Bucket name already taken; bucket not empty before delete	Use a unique bucket name; delete objects first
415	Unsupported Media Type	Source format not supported for this translation type	Check the APS supported formats list
429	Too Many Requests	Rate limit hit	Add backoff with `aps_retry()` and `Sys.sleep()`
500	Internal Server Error	Transient APS service error	Wait 30 s and retry; check APS status

23.2 Diagnosing Any Error

Before diving into the per-API sections, check the full error payload. APS returns descriptive messages in the response body:

```
resp <- someApsFunctionCall(...)

# Always check the status first
resp$status_code
#> [1] 401

# The body usually tells you exactly what's wrong
str(resp$content)
#> List of 3
#>  $ errorCode       : chr "AUTH-001"
#>  $ developerMessage: chr "The provided access token has expired."
#>  $ userMessage     : chr "Access denied."
```

To see the raw HTTP traffic, useful when the body is empty or unexpected, use httr2's verbose mode:

```
library(httr2)
# Wrap any request to trace the full exchange
req <- request("https://developer.api.autodesk.com/...") |>
  req_auth_bearer_token(myToken) |>
  req_verbose()
resp <- req_perform(req)
```

23.3 Installation & Setup

Problem: `library(AutoDeskR)` throws `there is no package called 'AutoDeskR'`.

Solution: Install the package first:

```
install.packages("AutoDeskR")         # stable CRAN release
# or
devtools::install_github("paulgovan/AutoDeskR")  # development version
```

Problem: Functions return errors about missing dependencies (`httr2`, `jsonlite`, `curl`).

Solution: These are required dependencies and should install automatically. If they didn't:

```r
install.packages(c("httr2", "jsonlite", "curl", "shiny"))
```

Problem: Some functions work but others return 403 or unexpected 404 errors.

Solution: In the APS Developer Portal, open your app's settings and confirm that **all required APIs are enabled**. Each API product (Data Management, Model Derivative, Design Automation, Reality Capture) must be explicitly added to the app. A 403 on a valid token almost always means the API isn't enabled for that app.

23.4 Authentication

Problem: `getToken()` returns a 401, or `myToken` is `NULL`.

Solution: Double-check that your `~/.Renviron` file is loaded and that the variable names match exactly:

```r
# Reload .Renviron without restarting R
readRenviron("~/.Renviron")

# Verify the values are present
nchar(Sys.getenv("client_id"))      # should be > 0
nchar(Sys.getenv("client_secret"))  # should be > 0
```

Also confirm the credentials belong to the correct app in the APS portal, a common mistake is having credentials from a sandbox app and trying to use a production bucket.

Problem: API calls that worked fine start returning 401 mid-session.

Solution: Tokens expire after exactly **3600 seconds** (1 hour). Re-call `getToken()` and update `myToken`:

```r
resp    <- getToken(id = Sys.getenv("client_id"),
                    secret = Sys.getenv("client_secret"),
                    scope  = "data:read data:write")
myToken <- resp$content$access_token
```

Problem: Getting 401 even with a fresh token.

Solution: The token scope probably doesn't match the API you're calling. Each task requires specific scopes — check the table in the Authentication chapter and make sure the scope string includes everything your call needs.

23.5 Data Management

Problem: `makeBucket()` returns 409 Conflict.

Solution: Bucket names are **globally unique across all APS applications**. Add a distinctive prefix:

```
# Bad - almost certainly taken
makeBucket(token = myToken, bucket = "test")

# Good - scoped to your org and project
makeBucket(token = myToken, bucket = "acmecorp-bridge-2026-dev")
```

Problem: `deleteBucket()` returns 409.

Solution: The bucket still has objects in it. Delete them first:

```
# List everything in the bucket
objects <- listObjects(token = myToken, bucket = "mybucket")

# Delete each object
for (key in objects$content$items$objectKey) {
  deleteObject(token = myToken, bucket = "mybucket", object = key)
}

# Now delete the empty bucket
deleteBucket(token = myToken, bucket = "mybucket")
```

Problem: `uploadFile()` fails or times out on large files.

Solution: `uploadFile()` is limited to 100 MB. Use `uploadFileSigned()` for larger files. It transfers via signed S3 URLs which don't have the same timeout constraints:

```r
resp <- uploadFileSigned(file = "large_model.rvt", token = myToken, bucket = "mybucket")
```

Problem: The encoded URN looks wrong or downstream functions can't find the object.

Solution: The raw `objectId` from `uploadFile()` must be Base-64 encoded before passing to Model Derivative or Viewer functions. Make sure you're encoding the right string:

```r
myUrn        <- resp$content$objectId        # raw URN from upload
myEncodedUrn <- jsonlite::base64_enc(myUrn)  # encoded URN for all subsequent calls
```

23.6 Model Derivative

Problem: `translateObj()` returns 415 Unsupported Media Type.

Solution: Not all file types can produce OBJ output. Supported sources include DWG, IPT, IAM, IPN, F3D, and FBX. Check the full list.

Problem: `checkFile()` stays in `"inprogress"` indefinitely.

Solution: Translation can take from seconds (simple DWGs) to many minutes (large Revit models). Wait at least 2× the expected time before giving up. If it's genuinely stuck, the job may have failed silently. Check `resp$content$status` for `"failed"` and look at `resp$content$progress` for an error message:

```r
resp <- checkFile(urn = myEncodedUrn, token = myToken)
resp$content$status
#> [1] "failed"
resp$content$progress
#> [1] "Translation failed: unsupported entity type on layer A-XREF"
```

Problem: `getOutputUrn()` returns an empty `derivatives` list.

Solution: The file hasn't finished translating yet, or the translation failed. Confirm `checkFile()` returns `status == "success"` before calling `getOutputUrn()`. Also confirm the `urn` you're passing is the *raw* (un-encoded) URN, not the encoded one. `getOutputUrn()` takes the raw URN.

Problem: `getData()` returns a very large response that takes a long time or crashes R.

Solution: Complex files can return hundreds of thousands of objects. Filter to the layers or object IDs you care about by inspecting `getObjectTree()` first and only fetching data for the relevant subset.

23.7 Design Automation

Problem: `makePdf()` returns 400 Bad Request.

Solution: Both the `source` and `destination` URLs must be **publicly accessible**. The AutoDesk cloud workers fetch and write these URLs directly. Common failures: - Private Google Drive share links (use "Anyone with the link" access) - `localhost` or `127.0.0.1` URLs (unreachable from the cloud) - Signed S3 URLs that have already expired

---

Problem: `checkPdf()` stays in `"pending"` for more than 5 minutes.

Solution: This usually means the job is queued behind other work items. Wait longer. Free-tier jobs can queue for 10–15 minutes at peak times. If it's still pending after 30 minutes, cancel and resubmit.

23.8 Reality Capture

Problem: The output mesh looks poor — holes, distorted geometry, missing areas.

Solution: Image quality is everything in photogrammetry. The most common causes of bad output: - **Too few images** — need at least 20, ideally 50+ - **Insufficient overlap** — every surface needs to appear in at least 3 photos from different angles - **Reflective or transparent surfaces** — glass and polished metal fool the matching algorithm - **Inconsistent lighting** — shots taken at different times of day with different shadows - **Camera movement instead of subject rotation** — move around the object, don't zoom from a fixed point

---

Problem: `waitForPhotoscene()` times out before the job finishes.

Solution: Increase the `timeout` parameter. A 500-image scene can take 90+ minutes:

```
done <- waitForPhotoscene(photoscene_id = myPhotosceneId,
                          token         = myToken,
                          interval      = 120,   # check every 2 minutes
                          timeout       = 7200)  # wait up to 2 hours
```

Problem: I downloaded the RCS file but `lidR` can't open it.

Solution: `lidR` reads LAS/LAZ, not AutoDesk's RCS format. Convert the file first using CloudCompare (free): *File* → *Open* → select the RCS → *File* → *Save As* → choose LAS format.

23.9 Viewer

Problem: The viewer opens but shows a blank canvas.

Solution: The file must be translated to **SVF** format before the viewer can display it. OBJ and STL translations won't work. Run `translateSvf()` and wait for `checkFile()` to return `status == "success"`, then reload the viewer.

Problem: The viewer throws a JavaScript authentication error.

Solution: The `data:read` scope is required. Other scopes (like `data:write`) are not enough on their own. Make sure `getToken()` was called with at least `scope = "data:read"`.

23.10 3D Geometry (rgl / Rvcg)

Problem: `rgl` window doesn't open in RStudio Server or a headless environment.

Solution: Set the null device before loading `rgl`:

```
options(rgl.useNULL = TRUE)
library(rgl)
# Now use rglwidget() to render to HTML instead of an OpenGL window
```

Problem: `vcgVolume()` returns zero or a nonsensical value.

Solution: `vcgVolume()` requires a **watertight (closed) mesh**. OBJ files translated from 2D DWG drawings are almost never watertight. Use the convex hull fallback from the Mesh Metrics chapter instead:

```
library(geometry)
pts  <- t(mesh_vcg$vb[1:3, ])
hull <- convhulln(pts, options = "FA")
hull$vol   # robust volume estimate for open meshes
```

Problem: `vcgImport()` fails with a parsing error on the OBJ file.

Solution: The Model Derivative API sometimes produces multi-part OBJ files with multiple `mtllib` references. Try importing with `rgl::readOBJ()` first to confirm the file is valid, or open it in a text editor to check for obvious corruption near the top of the file.

23.11 Digital Twins (Tandem)

Problem: Tandem API requests return 403 Forbidden even with a valid token.

Solution: The Tandem API requires an active **AutoDesk Tandem subscription**. It is not included in a standard APS free-tier account. Confirm the subscription is active in your AutoDesk account settings and that the app's Client ID is associated with a Tandem-enabled account.

Problem: `resp_body_json()` returns an empty list from the `/groups` endpoint.

Solution: The authenticated app doesn't have access to any Tandem facilities yet. Facilities must be explicitly shared with the app's Client ID through the Tandem web interface. Log into tandem.autodesk.com, open the facility settings, and add the app as a viewer or owner.

23.12 Getting More Help

- **APS community forum:** community.autodesk.com/t5/autodesk-platform-services — AutoDesk staff monitor this actively
- **AutoDeskR GitHub issues:** github.com/paulgovan/AutoDeskR/issues — for package-specific bugs
- **APS API status:** health.autodesk.com — check for ongoing incidents before spending time debugging

References

Adler, Daniel, Duncan Murdoch, et al. 2024. *Rgl: 3D Visualization Using OpenGL*. https://dmurdoch.github.io/rgl/.

Autodesk, Inc. 2024. "AutoDesk Tandem API — Developer Documentation." https://tandem.autodesk.com/api/v1/.

Govan, Paul. 2024. *AutoDeskR: An Interface to the AutoDesk Platform Services APIs*. https://github.com/paulgovan/AutoDeskR.

Iannone, Richard, Joe Cheng, Barret Schloerke, Ellis Hughes, Alexandra Lauer, JooYoung Seo, Ken Brevoort, and Olivier Roy. 2024. *Gt: Easily Create Presentation-Ready Display Tables*. https://gt.rstudio.com.

Morgan-Wall, Tyler. 2024. *Rayshader: Create Maps and Visualize Data in 2D and 3D*. https://www.rayshader.com.

Roussel, Jean-Romain, David Auty, Nicholas C. Coops, Piotr Tompalski, Tristan R. H. Goodbody, Alexis Sanchez Meador, Jean-François Bourdon, Florian de Boissieu, and Alexis Achim. 2020. "lidR: An r Package for Analysis of Airborne Laser Scanning (ALS) Data." *Remote Sensing of Environment* 251: 112061. https://doi.org/10.1016/j.rse.2020.112061.

Roussel, Jean-Romain, David C. Sterratt, et al. 2024. *Geometry: Mesh Generation and Surface Tessellation*. https://davidcsterratt.github.io/geometry/.

Schlager, Stefan. 2024. *Rvcg: Manipulations of Triangular Meshes Based on the 'VCGLIB' API*. https://CRAN.R-project.org/package=Rvcg.

Vanderkam, Dan, JJ Allaire, Jonathan Owen, Daniel Gromer, and Benoit Thieurmel. 2018. *Dygraphs: Interface to 'Dygraphs' Interactive Time Series Charting Library*. https://CRAN.R-project.org/package=dygraphs.

Wickham, Hadley. 2024. *Httr2: Perform HTTP Requests and Process the Responses*. https://httr2.r-lib.org.